European Sixth Framework Network of Excellence FP6-2004-IST-026854-NoE

Deliverable D6.5
Open Source Support and Joint Software Development

The EMANICS Consortium

Caisse des Dépôts et Consignations, CDC, France
Institut National de Recherche en Informatique et Automatique, INRIA, France
University of Twente, UT, The Netherlands
Imperial College, IC, UK
Jacobs University Bremen, JUB, Germany
KTH Royal Institute of Technology, KTH, Sweden
Oslo University College, HIO, Norway
Universitat Politecnica de Catalunya, UPC, Spain
University of Federal Armed Forces Munich, CETIM, Germany
Poznan Supercomputing and Networking Center, PSNC, Poland
University of Zürich, UniZH, Switzerland
Ludwig-Maximilian University Munich, LMU, Germany
University College London, UCL, UK
University of Pitesti, UniP, Romania

For more information on this document or the EMANICS Project, please contact:

Dr. Olivier Festor
Technopole de Nancy-Brabois - Campus scientifique
615, rue de Jardin Botanique - B.P. 101
F-54600 Villers Les Nancy Cedex
France
Phone: +33 383 59 30 66
Fax: +33 383 41 30 79
E-mail: <olivier.festor@loria.fr>

Document Control

Title:	Open Source Support and Joint Software Development
Type:	Public
Editor(s):	Olivier Festor
E-mail:	Olivier.Festor@loria.fr
Author(s):	WP6 Partners
Doc ID:	D6.5

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
0.1	2009-06-18	J. Schönwälder	Added YANG for Libsmi
0.2	2009-10-30	O. Festor and E. Nataf	Added JYANG/En suite integration
0.3	2009-12-06	B. Stelte and I. Hochstatter	Added iNagMon

Contents

1 Executive Summary

This deliverable presents the activity undertaken and results obtained within work-package 6 of the EMANICS network of excellence during the year 2009. The open call made for both packaging and development activities is presented and analyzed and for each of the 5 development activities as well as for the 2 packaging activities selected, the undertaken developments and software availabilities are presented.

2 Introduction

Open Source support, coherent joint software development and packaging is a very important initiative for ensuring strong integration, network survivability and transfer of technology developed within the network. In the management community, many Open Source components have been produced over the years, some of them being widely used today for several purposes. Recent evolutions in management and networking technologies however did slow the use of these very valuable software components and the rythm of new productions has decreased drastically. From the very beginning, EMANICS partners did recognize the need to support the Open Source initiatives in the area of management to foster their use and acceptance on large scale.

This effort was already initiated in the first phase of the network of excellence. The results obtained there are are reported in deliverable D6.2. In deliverable D6.3, we covered the activities undertaken from july 2007 to march 2008. D6.4 covers the april-december 2008 period. This deliverable covers the 2009 activity of WP6.

In December 2008, two calls have been issued to the EMANICS community: one for software development and one for Open Source software packaging. This call received a total of 14 development proposals together with 5 packaging initiatives. The work-package had a total of 60.000 Euros dedicated to these activities. The following proposals were selected and funded :

- Selected development initiatives :
  - Yang SMI (10K, JUB)
  - YangNetconf (10K, INRIA)
  - iNagMon (9K, UniBwM)
  - Sblomars (10K, UPC)
  - GridMonitoring (10K, UCL)
- Selected packaging initiatives:
  - Linubia (4 K UniZh)
  - LambdaMonitor (4K, PSNC)

To present the achievements, the report is structured as follows. Section 3 is dedicated to the LibSMI development for YANG support, first selected development project in phase 3 of EMANICS. Section 4 contains the description o the integration of two Open source components by INRIA : the YANG Java parser and the Ensuite Netconf framework. Section 5 presents a new Nagios monitoring front-end for the Iphone (INagMon). Section 6 is devoted to SBLOMARS, a distributed monitoring framework and load balancing engine for service clouds. The GridMonitoring proposal from UCL was canceled in late 2009 by the proposers. The associated ressources have been reallocated to the general activities of EMANICS

The two supported packaging activities are reported in section 7 and 8 (LINUBIA).

Section 9 concludes this deliverable. The text of the call issued in December 2009 as well as a summary of the proposals received are given as two appendices to this document.

3 YANG Language Parser for Libsmi

3.1 Introduction

YANG is a data modeling language used to model configuration and state data manipulated by the NETCONF protocol, NETCONF remote procedure calls, and NETCONF notifications. Today, the NETCONF protocol [1] lacks a standardized way to create data models. Instead, vendors are forced to use proprietary solutions. In order for NETCONF to be an interoperable protocol, models must be defined in a vendor-neutral way. YANG provides the language and rules for defining such models for use with NETCONF.

YANG is posed as simple and easy to learn because of its trivial and regular syntax. This language is a NETCONF-specific language designed to do what it needs to do well and it does not try to compete with XSD. YANG is fully modular and it enables development of certain parts of the model without influencing those parts that need no change. The YANG language is still under development and the latest draft version of the language was released on April 20, 2009

Libsmi is a C library which provides a convenient API for handling network management data model definitions written in various versions of the SMI. The project's main task is to create a YANG parser for the library, update its inner data structures to support the new language, and extend the API to support new features of the YANG language.

3.2 Motivation and problem statement

The YANG language is still being developed and it is not yet widely used. Availability of convenient programming tools which support this modeling language might facilitate its adoption. Libsmi is a portable C library that currently provides an API for access to various versions of the SMI. This library hides parsing details providing accessing SMI definitions for applications. Libsmi is a potential tool to be extended to support the YANG language.

When we started working on the project there was already an incomplete parser implementation for libsmi based on the early version of the YANG specification [2]. The implementation was incomplete and suffered from a number of limitations:

- the lexical analyzer was implemented inaccurately. There were various problems with line counting, string quoting, string concatenation and some general rules were defined wrong;
- the syntactic analyzer was implemented inaccurately. The implementation did not cover the whole grammar of the YANG language and there were many collisions with the specification;
- the implementation was incomplete: some language statements were stored to internal data structures, others were not;
- the implementation tried to retrofit the YANG data model into existing libsmi data structures and API functions. However it caused many problems, making the parser

much more complicated. It was confusing what functions one should use to handle a particular language definition, what data structure and members of the data structure correspond to the definition;

- there was no semantic analysis. References to groupings, types, features, and identities were not resolved, uniqueness of identifiers was not validated, modules knew nothing about definitions in their submodules.

The EMANICS funded development aimed at completing the parser for the YANG data modeling language in order to make it compliant with the latest version of the YANG language. At the time when the project was starting the available version of the specification was draft-ietf-netmod-yang-03 [3] and was used as a basis for the entire project.

3.3 Libsmi library

The core of the libsmi library allows management applications to access SMI MIB module definitions. The structure of the library [7] is internally divided into two layers. The upper layer provides the API to applications that make use of libsmi and internal data structures to management information in memory. The lower layer is a collection of drivers (actual parsers) that retrieve the management information from the SMI definitions. The details of the libsmi architecture are illustrated in the figure 1.

The libsmi library distribution also includes a collection of tools built on top of the library to check, analyze dump, convert, and compare MIB definitions.

In order to support a new modeling language by the libsmi library a driver for this language has to be implemented (a parser for the language), the way how to store language definitions in memory has to be specified by reusing already available data structures or implementing new ones, and the API to access module definitions has to be exposed. In order to be able to serialize module definitions stored in memory the smidump tool might be used which requires to implement a driver that defines serialization rules.

3.4 YANG module parsing

The parsing of a YANG module is accomplished over several phases, which includes lexical, syntactic and semantic analysis.

3.4.1 Lexical analysis

Lexical analysis allows to convert an input sequence of characters into a sequence of tokens which are atomic language units such as keywords, identifiers, text segment and number definitions. A lexical analyzer processes the input characters to categorize them according to function, giving them meaning.

In the project, lexical analysis is done by the lexical analyzer which is generated by the flex tool [5]. The flex program reads a description of a lexical analyzer to generate from

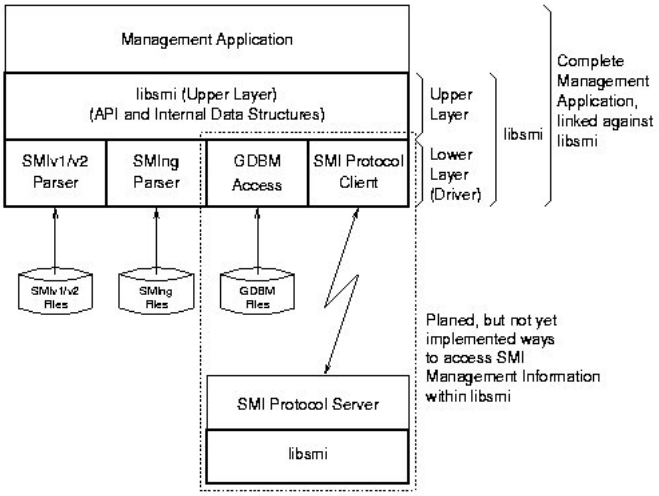

Figure 1: Libsmi architecture

the given input file. The description is in the form of pairs of regular expressions and C statements associated with them. Flex generates as output a C source file which is a lexical analyzer for the given description. This output file has to be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions and once it matches an input character sequence with a regular expression it executes the C code associated with it.

The file `lib/scanner-yang.l` contains the description of the lexical analyzer for the YANG language. A token in YANG is either a keyword, a string, ";", "{", or "}". First, regular expressions for all keywords of the YANG language are defined and C statements which return the type of the keyword associated with them. A string can be either unquoted or enclosed within double or single quotes, depending on the characters the string contains. Identifiers, identifier references (which are in the form prefix:identifier) and dates are specific kinds of strings. The rules to handle strings are defined in the description of the lexical analyzer. If a quoted string is followed by a plus character ("+"), followed by another quoted string, the two strings have to be concatenated, which is also accomplished here.

YANG supports two kinds of comments: a single line comment, which starts with "//" and ends at the end of the line; a block comment, which is enclosed within "/*" and "*/". The lexical analyzer contains rules which filter out both kinds of comments from the input character sequence.

Lexical analyzer is also responsible for counting line numbers to associate them with the tokens.

3.4.2 Syntactic analysis

Syntactic analysis processes a sequence of tokens obtained from the lexical analysis and determines their grammatical structure with respect to a given formal grammar. In other words, it checks whether it is possible to construct a particular sequence of tokens by

applying grammar rules of the language. Apart from the validation for correct syntax, the syntactic analysis also builds a parse tree for the input token sequence.

In order to create a syntactic analyzer for the YANG language, the bison parser generator [6] is used, which is a part of the GNU project. Bison converts a grammar description for a context-free grammar into a C program which can parse a sequence of tokens that conforms to that grammar.

The file `lib/parser-yang.y` is a Bison grammar file that contains the grammar of the YANG language in Bison syntax. The first part of this grammar file is Bison declarations, where the names for all terminal and nonterminal symbols are defined. The rest of the file implements the YANG ABNF grammar rules defined in the specification. In a Bison grammar, a grammar rule may have an action made up of C statements. Each time the parser recognizes a match for a rule, the action is executed. The associated with the grammar rules C statements accomplish simple validations and construct a representation of the parsing module in memory.

3.4.3 Semantic analysis

When the syntactic analysis is done, the YANG module is grammatically correct and the representation of the module in internal data structures is constructed, but it still does not mean that the module is valid. Semantic analysis is the phase in which semantic information is added to the tree, built in the previous phase. This phase performs semantic checks that require a complete module tree. For instance, these are some examples of the validations that have to be applied in this phase:

- resolve all references to user defined types, groupings, extensions, identities and features.
- check whether there are no circular dependencies between various module elements such as user-defined types, groupings, identities, features and modules themselves;
- check whether all identifiers have unique names in the scope of their namespaces;
- check whether restrictions applied to user defined types don't create collisions;

3.5 Data structures

The output of the syntactic analysis is stored in memory in internal libsmi data structures. The library has two layers of data structures - public and private. The public data structures are exposed through the library API. The private data structures are used only inside the library itself and contain the corresponding public data structures, as well as many additional fields, which are necessary in the validation phase.

The libsmi library already contains a set of implemented data structures for other modeling languages. While adding support for the YANG language there were two options:

- try to retrofit YANG model into existing data structures and APIs;

- create a separate set of data structures and API functions for YANG from scratch;

Each approach has its advantages and disadvantages. In the first case we might keep the API of the library unchanged and, theoretically, the applications, that use the libsmi library and have been developed before, don't require additional changes (or require minor changes). But, practically, it suffers from a number of limitations. The YANG language is pretty different from the languages libsmi supports and it would be not a trivial task to fit the YANG model into existing data structures and APIs. Even though it is doable, it would be necessary to modify some existing data structures and add new ones. It would result in very complicated data structures, some fields of which are used by one modeling language and not used by another one. As a result, the development process gets extremely difficult, because we have to operate on the data structures we have, but not ones that are best suited. In order to work with the library, one should spend much time on learning how the data is actually stored.

The later approach makes the development and usage of the library much simpler, but it loses compatibility. One can create data structures which are best suited for the language model. It will also make the API of the library clearer. In order to support the compatibility at the API level, it is possible to implement conversion functions which allow to convert a module between different languages.

In the project the second approach was chosen: all necessary data structures and the public API were defined from scratch.

3.5.1 Public data structures

Public data structures and user defined types for the YANG language are declared in the `lib/yang.h` file. All of them are named with the `Yang` prefix.

The `YangNode` data structure is used for representing YANG statements of any kind:

```
typedef struct YangNode {
    YangString      value;
    YangString      extra;
    YangDecl        nodeKind;
    YangStatus      status;
    YangConfig      config;
    YangString      *description;
    YangString      *reference;
} YangNode;
```

The `nodeKind` field of the `YangDecl` enumerated C type specifies the kind of the YANG statement that corresponds to the node. The `YangDecl` type declares 65 possible values (e.g. `YANG_DECL_IMPORT`). The `YANG_DECL_UNKNOWN` value represents the statements which are user defined extensions.

The `value` field of the `YangString` type holds the argument of the statement.

The `extra` field of the `YangString` type is only used by the statements which are user defined extensions with an argument.

The status field of the YangStatus type holds the value of the enclosed status statement (if the status is not present then the YANG_STATUS_DEFAULT_CURRENT value).

The config field of the YangConfig type holds the value of the enclosed config statement (if the config is not present then the YANG_CONFIG_DEFAULT_CURRENT value).

The description and reference fields of the YangString type hold the values of the enclosed description and reference statements respectively.

3.5.2 Private data structures

Private data structures are declared in the lib/yang-data.h file and have the _Yang prefix in their names.

All YANG module statements are represented in memory by the _YangNode data structure:

```
typedef struct _YangNode {
    YangNode            export;
    YangNodeType        nodeType;
    void                *info;
    int                 line;
    struct _YangTypeInfo *typeInfo;
    struct _YangNode   *firstChildPtr;
    struct _YangNode   *lastChildPtr;
    struct _YangNode   *nextSiblingPtr;
    struct _YangNode   *parentPtr;
    struct _YangNode   *modulePtr;
} _YangNode;
```

The first export field of the YangNode type is the corresponding public data structure of a YANG node, which is exposed through the API of the library. This fact allows to cast public data structure pointers to private data structure pointers, e.g.

```
_YangNode *nodePtr = (_YangNode*)yangNodePtr
```

where yangNodePtr is of the YangNode type.

The purpose of the nodeType field of the enumerated YangNodeType type is described later on in this report 3.8.4. All nodes corresponding to the definitions of the YANG module have YANG_NODE_ORIGINAL as the value of this field.

The info pointer is used to refer additional data structures, which hold the information specific for different types of YANG statements. For instance, if the node represents the module statement then the pointer contains a reference to the _YangModuleInfo data structure.

The line field holds the line number in the module file corresponding to the statement represented by the node.

The typeInfo field is intended to be used by nodes which represent the typedef statements (to hold a reference to the base type and restrictions applied to the type).

The _YangNode data structure is a list, every element of which has a reference to the next element nextSiblingPtr. The firstChildPtr and lastChildPtr fields hold pointers to the first and last nodes which represent statements enclosed into the current node. The parentPtr field holds a pointer to the node that represents the statement which encloses this one.

The modulePtr field is a reference to the node corresponding to the module statement to which the current node belongs.

Using the same data structure for representing all YANG statements makes it possible to process all nodes of a module tree in the same way, which allows to generalize the logic of many validations and ease the implementation.

3.6 YANG public API functions

At the present the public API for handling YANG module definitions is fairly poor. But all required data is stored in private data structures and later on another view on them may be exposed.

All YANG API functions have the yang prefix and are declared in the lib/yang.h file:

- `int yangIsModule(const char* name)`

  returns true if the module with the given name is a YANG module;

- `YangNode *yangGetModule(char *name)`

  returns a pointer to the struct YangNode for the module with the given name or NULL if the YANG module with the given name has not been loaded;

- `YangNode *yangGetFirstModule(void)`

  returns a pointer to the struct YangNode that represents the first loaded YANG module or NULL if no YANG module has been loaded;

- `YangNode *yangGetNextModule(`
  `YangNode *yangModulePtr)`

  returns a pointer to the struct YangNode that represents the YANG module loaded after the given module yangModulePtr or NULL if there is no YANG module loaded after it;

- `YangNode *yangGetFirstChildNode(`
  `YangNode *yangNodePtr)`

  returns a pointer to the struct YangNode that represents the first child node of the yangNodePtr node or NULL if that node does not have child nodes;

- `YangNode *yangGetNextSibling(`
  `YangNode *yangNodePtr)`

  returns a pointer to the struct YangNode that represents next sibling node of the yangNodePtr node or NULL if there is no next sibling node;

- `int yangIsTrueConf(YangConfig conf)`

  returns `1` if the given `YangConfig` value is true and `0` otherwise.

While working with the YANG API, there are several library API functions which still have to be used (for initialization and adjustment operations):

- `int smiInit(const char *tag)`

  initializes internal data structures;

- `void smiExit(void)`

  releases allocated resources;

- `char *smiLoadModule(const char *module)`

  loads a module with the given name in memory;

3.7 Implementation details

3.7.1 Module loading

YANG modules are loaded into memory by execution of the function

```
loadYangModule(const char *modulename,
               const char *revision,
               Parser *parserPtr),
```

where the module name must be specified, the revision and pointer to a parser are optional arguments. The `Parser` data structure declared in the `lib/data.h` file is used by all language drivers and contains a collection of parsing parameters and flags.

The execution of this function triggers the parsing of the module. While parsing the module, C statements associated with the grammar rules (defined in the Bison grammar description) are executed. These statements create the corresponding node in the module tree for every module definition and apply validations that don't require a complete module representation in memory.

In order to add a new node into the module tree the function

```
_YangNode *addYangNode(const char *value,
                       YangDecl nodeKind,
                       _YangNode *parentPtr)
```

is used. The first parameter `value` of the function is the argument of the YANG statement (e.g., the name of the module for the module statement, an XPath expression for the must, when and augment statements). The second parameter represents the type of the YANG statement that corresponds to the created node. And the last argument is a pointer to the parent node.

For some module tree nodes the `info` field of the `_YangNode` data structure is initialized here. What data is stored in this field depends on the type of the YANG statement:

”module” and ”submodule” statements the `info` field refers to the `_YangModuleInfo` data structure:

```
typedef struct _YangModuleInfo {
    char              *prefix;
    char              *version;
    char              *namespace;
    char              *organization;
    char              *contact;
    _YangParsingState parsingState;
    struct _YangNode  *originalModule;
    struct YangList   *submodules;
    struct YangList   *imports;
    void              *parser;
} _YangModuleInfo;
```

The `prefix`, `namespace`, `organization` and `contact` fields hold the arguments of the prefix, namespace, organization and contact statements, enclosed into the module or submodule. The `version` field contains the specified version of the YANG language.

The `parsingState` field of the enumerated `_YangParsingState` type represents the parsing state of the module and contains one of the following values:

- `YANG_PARSING_IN_PROGRESS` - the module is currently being parsed;
- `YANG_PARSING_DONE` - parsing of the module has already been finished;

This field is used to be able to determine circular dependencies between modules and submodules.

The `originalModule` field holds a pointer to the root of the unprocessed YANG module tree (before the tree has been expanded by instantiation of ”augment”, ”uses” statements).

The `submodules` and `imports` fields are linked lists of pointers to included submodules and imported modules respectively (YANG list data structures are examined later in this section).

”if-feature”, ”typedef”, ”uses”, ”base” statements and user-defined extensions the `info` field refers to the `_YangIdentifierRefInfo` data structure:

```
typedef struct _YangIdentifierRefInfo {
    char        *prefix;
    char        *identifierName;
    _YangNode   *resolvedNode;
    _YangNode   *marker;
} _YangIdentifierRefInfo;
```

These statements take an identifier reference as an argument. An identifier reference consists of the prefix of the module (which may be omitted if one refers the element in the current module) and the identifier, which are stored in the `prefix` and `identifierName` fields respectively. The `resolvedNode` field is a pointer to the node in the current or imported module corresponding to the identifier reference. The `marker` field is used to detect circular dependencies between module elements.

"grouping" statement the `info` field refers to the `_YangGroupingInfo` data structure:

```
typedef struct _YangGroupingInfo {
    _YangParsingState state;
} _YangGroupingInfo;
```

This data structure contains only one field, which represents the parsing state of the grouping definition and is used in circular dependency detection.

"key" and "unique" statements These statements take as an argument a string, which contains a space separated list of node identifier references. The `info` field holds a parsed list of identifier references.

3.7.2 Importing and including modules

YANG allows to specify the revision of the imported module or included submodule. When a module is published, it uses the current available revisions of other modules. Publishing new revisions of the imported modules does not affect the importing module, since it still uses revisions which have been published before. This YANG mechanism allows modules to evolve independently over time.

According to the specification, YANG modules and submodules are typically stored in files, one module or submodule per file, and the name of the file should be on the form:

```
(sub)module-name ['.' revision-date] ('.yang')
```

When the import (include) statement is encountered in the parsing module, the corresponding module (submodule) is loaded into memory by

```
_YangNode *externalModule(_YangNode *importNode);
```

function invocation. If the revision for an imported module is present, then the implementation tries to load a module with the specified revision. If such module has not been found, the library attempts to load a module with no revision. While loading imported modules (submodules), there is validation that allows to detect circular dependencies between them.

3.7.3 Private API functions

Private API functions are defined in the `lib/yang-data.h` file. Some of them have been already mentioned and explained in the previous sections. Here is a list of primary private API functions:

- ```
  _YangNode *loadYangModule(
                      const char *modulename,
                      const char * revision,
                      Parser *parserPtr)
  ```

  loads a module with the given name and revision.
  ```

- `void yangFreeData()`

  frees the memory allocated for all loaded YANG modules.

- `YangNode *findYangModuleByName(`
  `const char *modulename, char* revision)`

  returns a pointer to a module with the given name and revision. The `revision` parameter is optional here and `NULL` may be used if the revision is not needed.

- `_YangNode *findYangModuleByPrefix(`
  `_YangNode *module, const char *prefix)`

  returns a pointer to a module imported from the given module with the specified prefix.

- `_YangNode* findChildNodeByType(`
  `_YangNode *nodePtr, YangDecl nodeKind)`

  returns a pointer to the first child node of the given node by the specified node type.

- `_YangNode* findChildNodeByTypeAndValue(`
  `_YangNode *nodePtr, YangDecl nodeKind,`
  `char* argument)`

  returns a pointer to the first child node of the given node by the specified node type and argument value.

- `_YangNode* resolveReference(`
  `_YangNode *currentNodePtr,`
  `YangDecl nodeKind,`
  `char* prefix,`
  `char* identifierName)`

  resolves a reference to a node in the scope of the `currentNodePtr` node by the given node type (`nodeKind`), module prefix and argument value (`identifierName`). The `prefix` is an optional parameter and can be omitted if the referred node is within the current module. While resolving a reference the implementation searches up the levels of the hierarchy in the schema tree, starting at the `currentNodePtr` node. If the given prefix does not belong to the current module, the implementation searches for a node at the top level of the imported module with the specified prefix. If a node has not been found in the module (either the current or imported one), then the implementation searches in all submodules of the module.

- `YangModuleInfo *createModuleInfo(`
  `_YangNode *modulePtr)`

  returns a pointer to the `YangModuleInfo` structure created for the given module.

3.7.4 YANG list data structures

A linked list is an essential data structure in the YANG parser implementation. It is used to hold elements of various types. A generic linked list is defined in the `lib/yang-data.h` file as follows:

```
typedef struct YangList {
     void            *data;
     struct YangList *next;
} YangList;
```

The functions

```
YangList *addElementToList(
                     YangList *firstElement,
                     void *data)
```

and

```
YangList *addLastElementToList(
                     YangList *lastElement,
                     void *data)
```

allow to add elements to the beginning and end of the list.

In order to ease the usage of the lists, there is a set of functions with the `list` prefix, e.g.

```
_YangNode  *listNode(YangList *e),
```

which allow to avoid type casting of the the `data` field.

3.8 Semantic analysis

Syntactic analysis constructs a module tree and stores it in memory. Even though the grammar of the module is checked in this phase, there are still some kinds of validations that require a complete module tree and can not be accomplished here. Semantic analysis performs these validations as we already have the complete module stored in internal data structures.

All functions related semantic analysis are implemented in the file `lib/yang-check.c`. The function

```
void semanticAnalysis(_YangNode *module)
```

is the entry point of this phase. It takes as an argument a pointer to a module node that needs to be validated.

3.8.1 YANG extensions

YANG is an extensible language. A module can introduce YANG extensions by using the extension keyword. When an extension is used, the extension's keyword is qualified using the prefix with which the extension's module was imported:

```
module "extensions" {
    namespace "urn:smilib:params:ext";
    prefix ext;

    extension ext1;
    extension ext2 {
        argument param1;
    }
    ext:ext2 arg1 {
        ext:ext1;
    }
}
```

All user defined extensions have to be resolved and validated. The argument of the extension has to be present only if it is declared in the extension's definition.

3.8.2 Resolving references

The "if-feature", "type", "uses" and "base" YANG statements' argument refers to the "feature", "typedef" (if a built-in type is not used), "grouping" and "identity" nodes in the module tree respectively. All these references have to be resolved.

These statements can refer the definitions either within the current module or one of the imported ones (all imported modules have to be loaded in memory before the semantic analysis phase). A module can be divided into a number of submodules, but the external view of the module remains as a single module, regardless of the presence or size of its submodules. Therefore, while resolving a reference in a certain module, if a referred node has not been found there, one should search for it in all submodules of the module. All these statements must not reference themselves, neither directly nor indirectly through a chain of other statements.

The references are resolved in the function

```
void resolveReferences(_YangNode* node)
```

implemented in the `yang-check.c` file. The parameter is a pointer to the root node of a module subtree for which references need to be resolved.

The function resolves references to the "feature", "typedef", "grouping", "base" and "extension" nodes. When the reference is resolved the pointer to the referred node is stored to the `resolvedNode` field of the `_YangIdentifierRefInfo` structure (for these kinds of nodes this data structure is used as the `info` field of the `_YangNode`). The function also validates whether there are no circular dependencies between these statements. The usage of the same data structure for representing all YANG statements in a module tree allows to generalize the code of the function for different kinds of statements.

3.8.3 ”uses” statement

The ”uses” statements reference ”grouping” definitions. The effect of a ”uses” reference is that the nodes defined by the grouping are copied into the current schema tree, and then updated according to the refinement statements.

This statement changes the module's schema tree and this may result in a number of errors that should be additionally validated, e.g:

```
module grouping
{
    namespace "urn:smilib:grouping";
    prefix g;

    leaf leaf1 {
        type string;
    }
    uses grouping1;
    grouping grouping1 {
        leaf leaf1 {
            type int8;
        }
    }
}
```

after the ”uses” statement is instantiated the module looks as follows:

```
module grouping
{
    namespace "urn:smilib:grouping";
    prefix g;

    leaf leaf1 {
        type string;
    }
    leaf leaf1 {
        type int8;
    }
    grouping grouping1 {
        leaf leaf1 {
            type int8;
        }
    }
}
```

where we have two leafs with the same name at the top level of the module, which is not allowed. The uniqueness of identifiers is not the only thing that may be broken (there are some other statements, validation of which depends on where they are in the module tree, e.g., validation of the ”config” statement described below in this section).

The ”uses” statement may refine some of the properties of each node in the grouping with the ”refine” substatement. The argument of this statement is a string which identifies a node in the grouping. A set of properties which may be refined depends on the type of the target node in the grouping.

Probably the easiest way to handle ”uses” statements is to instantiate them by coping nodes defined in the grouping and afterwards refine them, according to the ”refine” statements. This allows not to change the general validation logic of the parser. The instantiation is performed by the function

```
int expandGroupings(_YangNode *node),
```

which takes a pointer to the root node of a module subtree as an argument and processes all ”uses” statements in that subtree. The function returns `0` if a circular dependency between groupings has been found while instantiating, and `1` otherwise. If a grouping itself contains ”uses” statements they are instantiated before the grouping's instantiation.

For every ”refine” statement the target node is resolved, and the refinement is accomplished if it's allowed, depending on the type of the target node.

3.8.4 ”augment” statement

YANG allows a module to insert additional nodes into data models, including both the current module (and its submodules) or an external module. The argument of the ”augment” statement defines the location in the data model hierarchy where new nodes are inserted. In the same way as the ”uses” statement, the ”augment” statement changes the module's schema tree, what requires some additional validations.

The function

```
void expangAugments(_YangNode* node)
```

processes all ”augment” statements in a subtree given by the pointer to the root node. For every ”augment” statement in a subtree the target node is resolved and new child nodes are added to it (this process involves a number of various checks), e.g.

```
submodule augment {
    belongs-to augment-super {
        prefix "as";
    }
    container interfaces {
        container entry {}
    }
    augment "/as:interfaces/as:entry" {
        leaf ifIndex {
            type int8;
        }
    }
}
```

when the ”augment” statement of the submodule is processed it looks in the following way:

```
submodule augment {
    belongs-to augment-super {
        prefix "as";
    }
    container interfaces {
        container entry {
            leaf ifIndex {
                type int8;
            }
        }
    }
    augment "/as:interfaces/as:entry" {
        leaf ifIndex {
                type int8;
        }
    }
}
```

The function

```
_YangNode *resolveXPath(
            _YangNode *nodePtr)
```

finds the target node for the ”augment” statement. It takes a node that corresponds to a YANG statement with an argument in the absolute-schema-nodeid or descendant-schema-nodeid form (the augment’s argument is given in one of these forms) and returns a pointer to the node in the current or imported module referenced by this argument.

The function

```
void copySubtree(
          _YangNode *destPtr,
          _YangNode *subtreePtr,
          YangNodeType nodeType,
          int skipMandatory)
```

is used to instantiate ”uses” and ”augment” statements. It copies a subtree given by the pointer `subtreePtr` to the destination node `destPtr`. The `nodeType` argument of the enumerated `YangNodeType` type may have one of the following values:

- `YANG_NODE_EXPANDED_AUGMENT`- the subtree is created by the ”augment” statement;
- `YANG_NODE_EXPANDED_USES` - the subtree is created by the ”uses” statement.

While coping the subtree, the value of the `nodeType` argument is stored to the `nodeType` field of the copied `_YangNode` structures, so that it is possible to distinguish original nodes from the ones that have been added to the module tree by instantiation of some statements. The `skipMandatory` argument says whether mandatory nodes should to be copied.

3.8.5 Identifier uniqueness

Identifiers are used to identify different kinds of YANG items by name. Each identifier is valid in a namespace which depends on the type of the YANG item. The namespaces for the different types of identifiers are defined in section 6.2.1 of the specification [4]. For the grouping and typedef statements there is an additional restriction: if the grouping or typedef is defined at the top level of a YANG module or submodule, their identifiers must be unique within the module (in the scope of the grouping and typedef identifier namespaces respectively).

Identifier uniqueness has to be validated after the ”uses” and ”augment” statements have been instantiated. The function

```
void uniqueNames(_YangNode* nodePtr).
```

performs this validation. It traverses the module tree and for every node executes the function

```
int validateNodeUniqueness(_YangNode *nodePtr)
```

which accomplishes the actual validation of the identifier uniqueness for the given node.

3.8.6 ”config” statement

According to the specification [4], if a node has ”config” ”false”, then no node underneath it can have ”config” set to ”true”. This condition is validated by the function

```
void validateConfigProperties(
                    _YangNode *nodePtr,
                    int isConfigTrue),
```

where `nodePtr` is a pointer to the node to validate and `isConfigTrue` is `1` if the current config value is true and `0` otherwise. In order to run this validation on a module we need to pass a pointer to the module and `1` as arguments of the function.

3.8.7 Other validations

In addition to the validations mentioned before, there are a lot of others, which are specific for different types of YANG language definitions.

For instance, here is a list of validations that have to be applied to the list definitions:

- the ”key” statement must be present if the list represents configuration;
- a leaf identifier must not appear more than once in the key;
- each key’s leaf identifier must refer to a child leaf of the list;

- a leaf that is part of the key can must not be the built-in type ”empty”;
- all key leafs in a list must have the same value for their ”config” as the list itself;
- each schema node identifier of the ”unique” statement must refer to a leaf;
- if one of the referenced by the ”unique” statement leafs represents configuration data, then all of the referenced leafs must represent configuration data;

3.8.8 Not implemented validations

There are still several validations that have to be implemented in the future:

- YANG provides a number of various restrictions for types. At the present these restrictions mainly are not checked in the implementation. For instance, we can apply the ”range” restriction to the string type or restrictions can contradict each other.
- the ”must” and ”when” statements take as an argument a string which contains an XPath expression. These XPath expressions are not validated.
- The YANG ”deviation” statement allows to specify what part of the module hierarchy the device does not implement faithfully. This statement may change the schema tree by adding, removing and renaming properties. Therefor additional checks are required to make sure that correctness of the module has been broken by applying deviations.
- If a definition is ”current” (the status of the definition is ”curent”) then it must not reference a ”deprecated” or ”obsolete” definition within the same module. If a definition is ”deprecated”, it must not reference an ”obsolete” definition within the same module.

3.9 Smidump driver

Smidump is a program which comes with the libsmi library distribution and allows to dump the content of a MIB module. Currently smidump has several drivers for dumping SMI structures in different language formats. In the course of the project the YANG driver for smidump has been implemented. It allows to print a YANG module stored in memory back to text format YANG definitions. Since the YANG language definitions are stored in data structures which are not used by other languages and there is no converter from any other format to YANG, currently the driver is not very useful. But it may be employed for testing of the YANG parser.

The file `tool/dump-yang-sk.c` contains the implementation of the driver, which traverses the entire module node hierarchy and serializes all nodes. The driver is registered with the `yang-sk` name.

3.10 Testing

From the very beginning of the project, while implementing a new grammar rule or check, we have been trying to create a test case that allows to validate that rule or check. Every test case consists of two files: an input file that contains YANG module definitions and the file with the expected output for the test case. In order to test the library on a certain test the smilint tool has to be executed on the test input file, and the obtained output file has to be compared to the expected one. All test files are available in the folder `test/yang` of the library infrastructure.

In order to automate the process of running all test cases at once, there is a script available at `test/parser-yang.test`. Having changed the source code of the YANG parser, now it is easy to validate whether something has been broken by simple automated running of all test cases.

3.11 Related work

Pyang [8] is a YANG validator, transformator and code generator, written in python. It can be used to validate YANG modules for correctness, to transform YANG modules into other formats such as YIN, DSDL and XSD, and generate code from the module definitions. While working on the project we have been constantly using that tool to clarify vague parts of the YANG specification. At the same time we have been testing pyang and have found a number of issues, which we reported to the pyang bug tracking system.

Here is a list of all reported by me issues:

- `http://code.google.com/p/pyang/issues/detail?id=5`
- `http://code.google.com/p/pyang/issues/detail?id=6`
- `http://code.google.com/p/pyang/issues/detail?id=7`
- `http://code.google.com/p/pyang/issues/detail?id=8`
- `http://code.google.com/p/pyang/issues/detail?id=9`
- `http://code.google.com/p/pyang/issues/detail?id=10`
- `http://code.google.com/p/pyang/issues/detail?id=11`
- `http://code.google.com/p/pyang/issues/detail?id=12`
- `http://code.google.com/p/pyang/issues/detail?id=13`
- `http://code.google.com/p/pyang/issues/detail?id=14`
- `http://code.google.com/p/pyang/issues/detail?id=15`
- `http://code.google.com/p/pyang/issues/detail?id=17`
- `http://code.google.com/p/pyang/issues/detail?id=18`

3.12 Conclusions

In the course of the project a YANG parser has been implemented. The parsing of a YANG module is accomplished over several phases: lexical analysis converts an input sequence of characters of the YANG module into a sequence of atomic language units (tokens); syntactic analysis determines whether it is possible to get the sequence of tokens obtained from the previous step by applying rules of the YANG language grammar, and constructs a module representation in memory; and semantic analysis performs validations which require a complete module tree. As of now the first two phases and most of the semantic analysis validations have been completed. Even though the parser validates most of the YANG language definitions, there are still some left (see section 3.8.8) that have to be implemented in the future.

While working on the project, a number of test cases have been collected for the parser. There is a script available that allows to run all test cases automatically.

A driver for smidump has been implemented that allows to print a YANG module stored in memory back to text format YANG definitions.

References

[1] R. Enns. RFC4741 NETCONF Configuration Protocol. Technical report, Network Working Group., December 2006.

[2] M. Bjorklund. YANG - A data modeling language for NETCONF draft-ietf-netmod-yang-02. Technical report, Network Working Group. February 2008.

[3] M. Bjorklund. YANG - A data modeling language for NETCONF draft-ietf-netmod-yang-03. Technical report, Network Working Group. February 2009.

[4] M. Bjorklund. YANG - A data modeling language for NETCONF draft-ietf-netmod-yang-05. Technical report, Network Working Group. April 2009.

[5] flex - The fast lexical analyser. `http://flex.sourceforge.net/`, May 2009.

[6] Bison - GNU parser generator. `http://www.gnu.org/software/bison/`, May 2009.

[7] libsmi - A library to access SMI MIB Information. `http://www.ibr.cs.tu-bs.de/projects/libsmi/`, May 2009.

[8] pyang - An extensible YANG validator and converter in python. `http://code.google.com/p/pyang/`, May 2009.

4 Yang and ENSUITE integration

4.1 Introduction

Configuration management importance is increasing with the growing size and the complexity of network resources. In the Internet context, the netconf working group has proposed the NETCONF [1] protocol as a standard to manage configuration of network devices. This protocol is tailored to configuration operation i.e. setting and/or getting configuration data values to/from devices. Values are transmitted in a XML document. The standardization body admits this should be improved by a data model that will give semantic to these data values and should be used as a contract between device vendors and application developers.

YANG [2] is the data modeling language proposed by the netmod working group. YANG can be compared to SMI in the SNMP framework because it is a data modeling language and because data values are distributed and accessible through a protocol. With YANG one can specify complex but human-readable configurations for any network device. YANG is presented as a more focused and adapted data modeling language for configuration management than XML Schema or Relax NG [5,6]. On the server side, any vendor can use such specifications to build a NETCONF server that will maintain the local configuration objects that map YANG specifications. On the client side, configuration applications need data values to operate, test or browse configurations. YANG data model specifications are a formal contract between devices' vendors and configuration applications and our goal is to provide tools helping users to ensure that the contract is respected.

The objective of this contribution is to demonstrate the feasibility of an End-to-End YANG-aware management framework and to describe how it can be implemented in an open source framework.

4.2 jYang back-end

jYang [3, 4] is an open source parser for the YANG language. It is written in java with the javaCC library [5] A jYang compilation starts with one or more YANG files references that will be loaded by the parser. All import and include statements are followed without parsing twice the same file. The internal representation of YANG specifications is one of the output of jYang and is composed of the java objects tree we show in slide 3.

Another useful output of jYang is the YangTreeNode that is also a java object tree where each node contains a reference to a YANG statement. This tree represents YANG specification but where typedef and grouping are copied at places where they are used. YangTreeNode is an intermediary representation of YANG specifications between static data model and data values on the wire. Its goal is to alleviate the work of any backend that focuses on NETCONF data values. As an example on figure 2 the module b only contains a grouping definition that is used two times in the module a so the YangTreeNode from a that we note a, contains two YangTreeNode b from b (we do not show sa1 for readability of the picture).

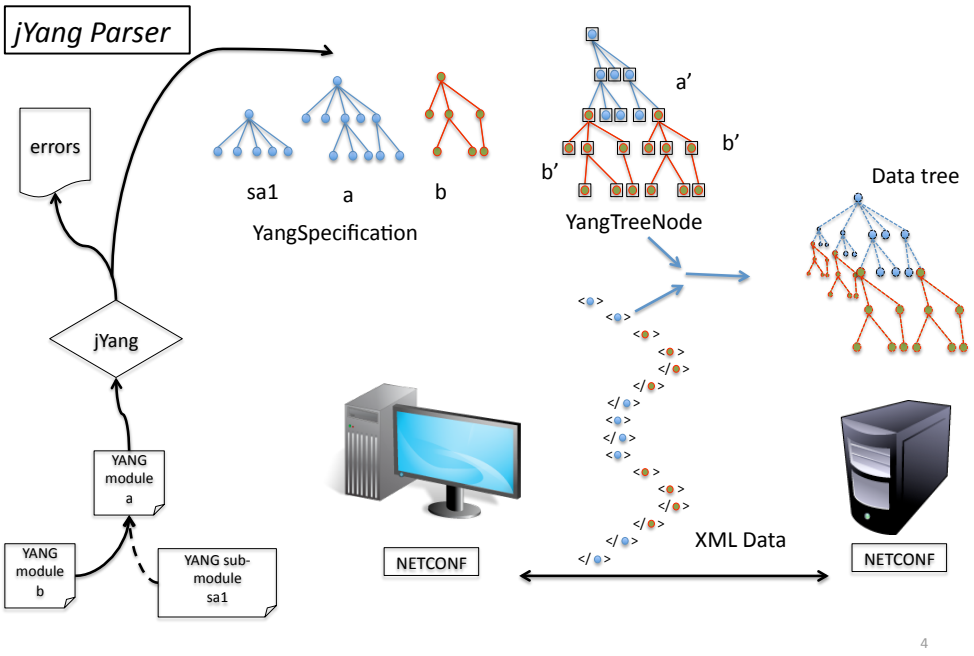

Figure 2: jYang back-end

The YangTreeNode is used to produce the YANG standard data tree from XML data of NETCONF operations. The figure 2 suggests that the data tree has more nodes than the YangTreeNode and that these nodes have a common pattern. This can be the case when a YANG list (or leaf-list) is defined because the data tree will contain each entry of the list (or each value of a leaf list). At the opposite if a specification is made with plenty of choice statements then the data tree will only show one of the cases from the NETCONF data values.

4.3 ENSUITE framework opening

YencaP Manager is an open source NETCONF application that can send queries and receive responses with any NETCONF server. The NETCONF Client can manage several NETCONF sessions with different servers at one time. Each of theses sessions is initialized from the HTTPS server inside the YencaP Manager when a user opens a HTTPS session. There is a one to one mapping between HTTPS and NETCONF sessions even if two users are accessing the same NETCONF server. The couple (YencaP / YencaP Manager) forms the ENSUITE framework [6].

As shown in figure 3, we had to extend YencaP to announce which YANG modules it implements (together with version and revision information) as a capability in its standard hello message. On the client side, a YANG loader is used by the YencaP Manager when such a capability is detected. We do not constraint the YencaP Manager to only work with YANG but to accept servers that are YANG enabled or not. The YANG loader gets

the specifications from an external repository and builds a specific YangTreeNode for the data model maintained by the server. The YANG loader is a java program that uses jYang to dynamically parse YANG specifications. We took this choice because we suppose the YencaP Manager will discover servers without knowledge of their configuration and thus must be able to dynamically load and parse any YANG specification. There is also the creation of glue parts in the YangTreeNode as a virtual netconf container, that are needed to built one YangTreeNode from several YANG modules. The YANG specification repository is shown as an external element of the YencaP Manager as it should be a global repository implemented for example as a web service.

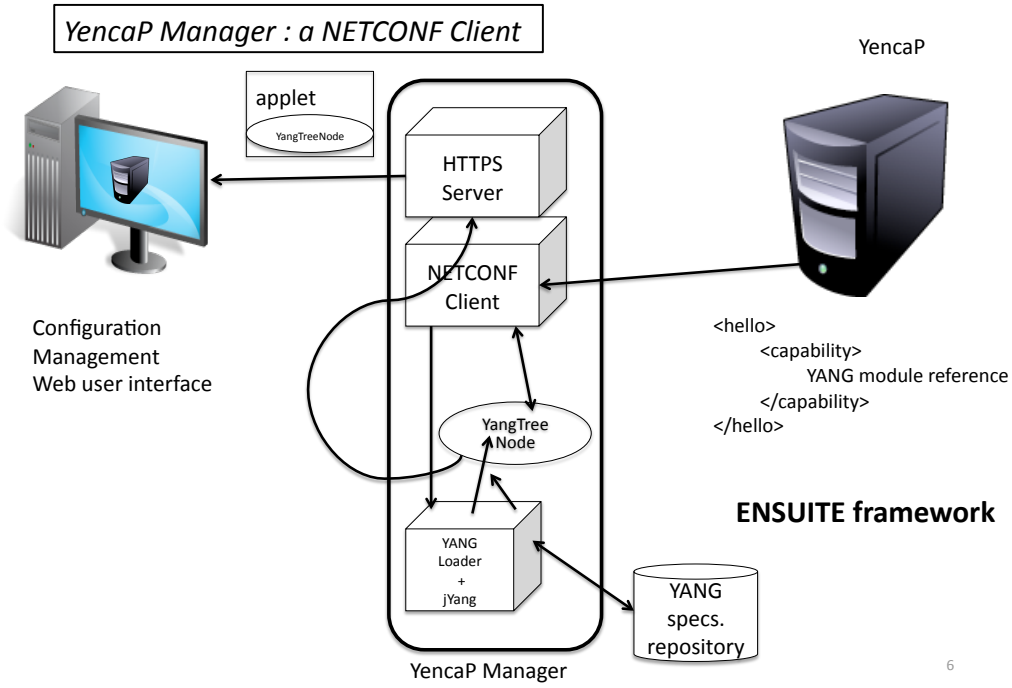

Figure 3: ENSUITE and jYang

Once a HTTPS session is opened the user can ask for the configuration of a YANG enabled device. In doing so it receives a java applet that contains the YangTreeNode for this server only. The applet will be loaded by the web interface to provide the user with a graphical interface representing the configuration.

4.4 Graphical Interface

On the upper left corner, the figure 4 shows the applet part of the web interface, the user gets when asking for the configuration of a device. This first view can be used as a YANG specification browser that looks like a file system browser (we use the swing Jtree interface). The tree view matches well with YANG because it defines a schema tree. Specific icons are used to distinct nodes, here NETCONF, network and interfaces are all YANG containers, interface is a YANG list and name, mac-address, mtu are YANG

leaves. A YANG list can have some key inside its leaf as is the name leaf referenced inside brackets in the interface list and by a little star on its leaf icon.

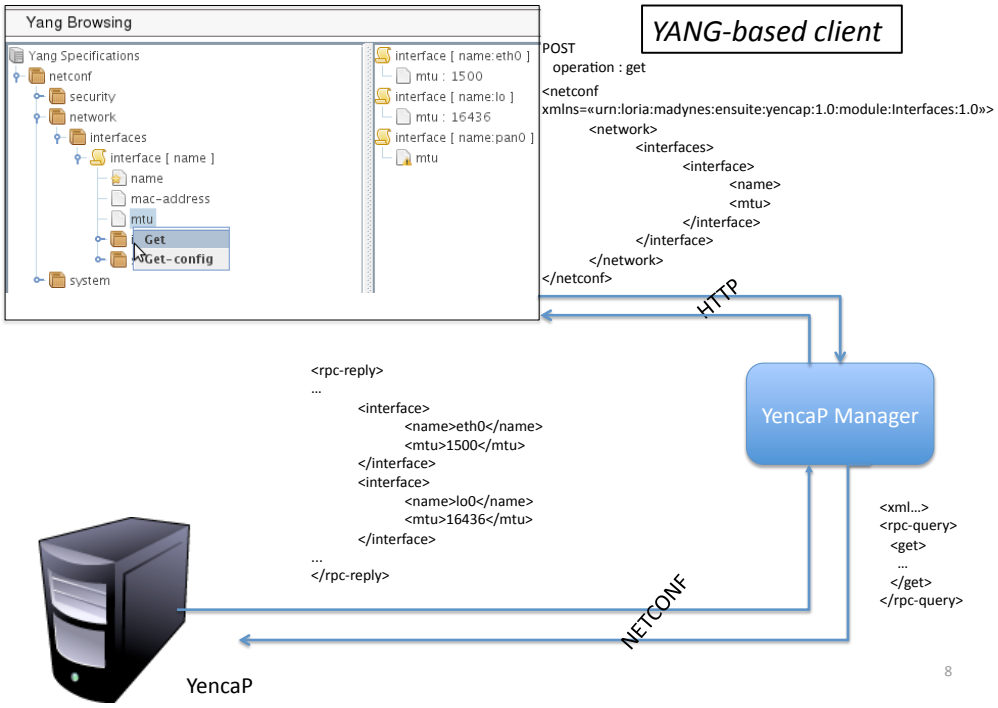

Figure 4: Yang GUI

From the GUI one can request the indirectly connected NETCONF device by a mouse contextual menu that pops-up when the right button is pressed on a YANG data type. When one of the standard NETCONF operations is chosen, the request is built from the root node (here the netconf virtual container) to the tree position of the selected node. The resulting XML document is sent inside a HTTP POST request. A specific header called operation is used to specify which NETCONF operation must be performed on the server (get, get-config or edit-config). Note that the key of the list is added to the request while it is not explicitly asked. This is an optimization because subsequent requests on lists (and especially on list entries) will likely need the key.

When the request is received by the YencaP Manager, the latter adds the rpc and filter mechanisms to surround the XML document received and sends a valid NETCONF request. From this step we are independent of any YANG concern because we are in a full NETCONF session. So the reply is sent by the NETCONF server to the YencaP Manager and this finish the NETCONF operation. Following is simply a cleaning of NETCONF XML data until the first node of the Data Store and its forwarding to the client applet that is waiting for the response. Figure 4 shows the response on the right part of the management applet. The request is synchronous because even if one request contains several data (as can be a request on a list) all of them are returned by one response. Note we have made our protocol synchronous on top of HTTP with several asynchronous requests. We plan to allow multiple selections for the same NETCONF operation to give access to the separate part of the NETCONF Data Store in one request.

The figure 5 depicts some functionalities of the Yang Browsing client. Part (a) is a simple access to a leaf in a container and where we get the response of a get-config operation. Part (b) shows that when editing a container, its components are listed with a warning until a correct value is given. Part (c) is an edit-config for a list. Here a list is edited entry by entry (that we call a list occurrence) and one can see an empty list entry ready to be filled. Note that a red mark is on the login leaf because it is the key of the list and so its value must be set. Part (d) illustrates the choice representation and its edition. A choice case (user-localization or nolocation) can only be edited if all of its components are set. On the same part the leaf called password is read marked because the value is not long enough. This is an example of dynamic constraint one can check with the tool. Range values of integer or float, pattern matching of string are also checked.

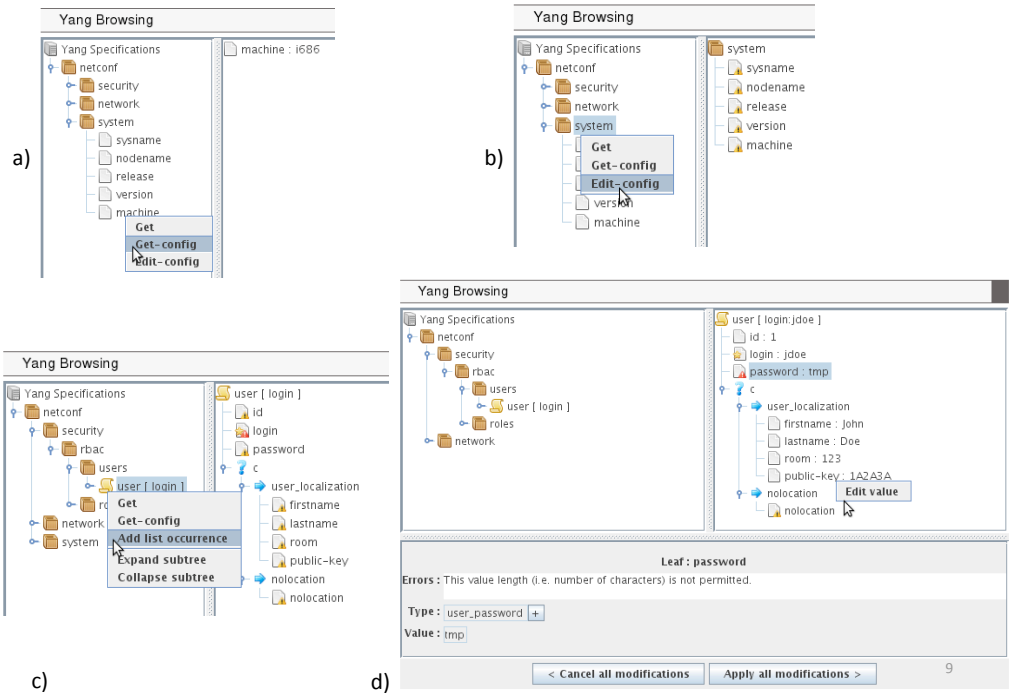

Figure 5: Yang GUI examples

4.5 Conclusion

Thans to this EMANICS funded project, we were able to provide two contributions to the network configuration domain. The first one is a YANG parser and semantic checker close to the actual version of the draft definition of YANG. The second contribution is the support within the ENSUITE framework of YANG based models both on the server and the client side.

We plan to extend YencaP to be able to generate parts of its code from YANG specifications, or build generic parts, to ensure the server maintains a valid Data Store compliant

with YANG . The server has to be able to send notifications especially those defined in YANG and must also accept user defined operations as there are YANG rpc and notification statements to do this.

We are also interested by all YANG constraints one can specify. Default value, must and presence conditions, references between values, length or pattern matching are some examples of such constraints. If one can have a NETCONF server with such knowledge this server will be enabled to autonomously checks its configuration. At the client side the constraints can ensure the manager does not make mistakes in its configuration operations and can notify users if constraints are not respected. This activity is left for future work around the platform.

References

[1] R. Enns. RFC4741 NETCONF Configuration Protocol. Technical report, Network Working Group., December 2006.

[2] M. Bjorklund. YANG - A data modeling language for NETCONF draft-ietf-netmod-yang-07. Technical report, Network Working Group. July 2009.

[3] jYang - An YANG validator and converter in java. `http://jyang.gforge.inria.fr/`, May 2009.

[4] E. Nataf, O. Festor. jYang a YANG parser in java. `http://hal.inria.fr/inria-00411261/en/`. August 2009.

[5] javaCC - Java Compiler Compiler. `https://javacc.dev.java.net/`.

[6] ENSUITE - a Netconf Framework. `http://ensuite.sourceforge.net`.

5 iNagMon - Network Monitoring on the iPhone

5.1 Introduction

Network monitoring is an elementary activity in network management as the gathered data is basis for all further action from troubleshooting problems to planning network extensions [1]. Commonly monitored information includes availability, load in the network and on hosts, response time and many more. On a high level of abstraction this information is often aggregated to show the status of a component or service in the simple indicator of a traffic light: green meaning that everything is working correctly, yellow stating that performance is somewhat restricted, and red reporting a failure.

A great number of tools for network monitoring exist ranging from open-source projects to commercial software and from general purpose to specialized systems [2, 3]. Their features vary similarly; most support standard protocols like snmp, facilitate distributed monitoring, and provide charts. Many tools also retain easy extensibility through the use of plugins and/or scripting. Like in many other institutions and companies, we use Nagios in our department to survey the status of network and services.

Nagios is extensible with add-ons that exist for many tasks. For our project, the NagVis (see Section 5.2.1) plugin is most relevant as it is a visualization add-on for Nagios. As network administrators are often not at their desk and want to be able to monitor the network when en route and also often when out of the office, providing visualization of Nagios data on mobile devices is the main goal of this work. More specifically, we focus on the development of an application for the Apple iPhone to display monitoring data. Figure 6 depicts the start screen of the iNagMon application that has been developed in this context. Here, the status information is given for each component in the monitored network.

5.2 Network Monitoring with Nagios

5.2.1 NagVis

NagVis is a visualization add-on for the well known network management system Nagios [5].

NagVis can be used to visualize Nagios data, e.g. to display IT processes like a mail system or a network infrastructure. Using data supplied by NDO (a Nagios add-on) it will update objects placed on maps in certain intervals to reflect the current status. These maps allow to arrange the objects to display them in different layouts:

- physical (e.g. all hosts in a rack/room/department)
- logical (e.g. all application servers)
- geographical (e.g. all hosts in a country)
- business processes (e.g. all hosts/services involved in a process)

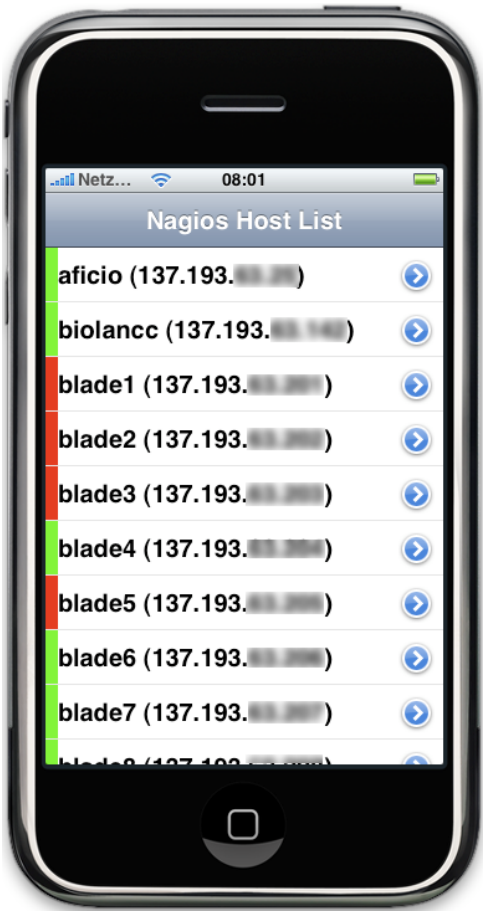

Figure 6: Start screen of the iNagMon application

In general NagVis is a presentation tool for the information which is gathered by Nagios. NagVis collects the information from back ends. There is a default backend delivered with NagVis: NDO MySQL Backend. This backend reads all information from the NDO (Nagios Data Out) MySQL database. The NDO is an add-on for Nagios. You can find it on the official Nagios Homepage.

You can add all objects from Nagios (Host, Services, Host groups, Service groups) on so called maps. Each map can be configured through its own configuration file. You can edit the configuration files directly by using your favorite text editor or the web configuration tool called WUI. Furthermore you can add some special NagVis objects to the maps. These objects are shapes, text boxes and reference objects for other maps.

Each of the objects on your maps can be configured to fit your needs. For example there are links to the Nagios frontend on each object which represents a Nagios object. You can easily customize these links. There is a hover menu which is enabled by default. The hover menu displays detailed information for each object. Hover menus can easily be modified by changing the templates for them. You can also disable the hover menu. By default the objects are displayed as icons on the map. You can change these icons by adding icon sets from the NagVis homepage or create your own. The objects can also be displayed as lines or as gadgets.

In addition to the normal maps there is an automap. The automap is generated automat-

ically based on the Nagios configuration. For using the automap you have to set up the parent relationships in Nagios. The automap generates a background image based on the configuration, the layout parameters and the parent relationships The NagVis add-on is great for the visualization of events on a full-screen device but the handling on a small device like the iPhone is worse.

5.2.2 iNagios - the iPhone web interface to Nagios

iNagios is a Web-Application that provides a simple read-only interface to a Nagios monitoring server that is optimized for use on an iPhone or iPod Touch [6]. It includes the familiar tactical overview, host and service detail views, host and service problem views as well as some reporting. In contrast to the here discussed application the iNagios web application needs a permanent network connection to the Internet in order to give a visualization of monitored Nagios events.

5.3 iNagMon Design and Implementation

5.3.1 NDO - Migration to SQLite

An important Nagios module is Nagios Data Output Utils (short NDO-utils). This module allows Nagios administrators to persist host and service configuration and host and service checks results to a relational database like MySQL. The module is parted in the NDOMOD event broker module and the NDO2DB database interface.
In the Nagios monitoring system the Nagios event broker provides information which the NDOMOD module makes available to remote clients like the NDO2DB interface. This interface communicates of a TCP socket with the NDOMOD and populates the database with new information. Figure 7 shows the typical data handover between the involved parties.

Unfortunately, the mobile device can not easily communicate with the database. The problem here lies in its mobile character of not having a permanent connection to the network. We can proceed from the assumption that in our case the iPhone user will cope often with network connectivity like long packet delays, packet lost, fluctuating bandwidth and especially network type switching (between GSM, UMTS, WLAN). After switching to another network type the connection is available again, but the connection to the database has to be resumed because of a new assigned IP address. In the time where no network connection is available the connection to the database is not possible, so the application is out of information.
Secondly a permanent data-connectivity increases the amount of transfered data which could be costly in a mobile environment. We need a solution where information is stored on the device and updated depending on network conditions.
In our concept we will therefore reduce the information content to a minimum and secondly use a small database on the device to overcome the mentioned problems. The used database schema (Figure 8) we have used for our application covers the four tables *host, hoststatus, services,* and *servicestatus* which we have adapted from the NDO-utils database. This reduced information content is the base for our database. In order to get

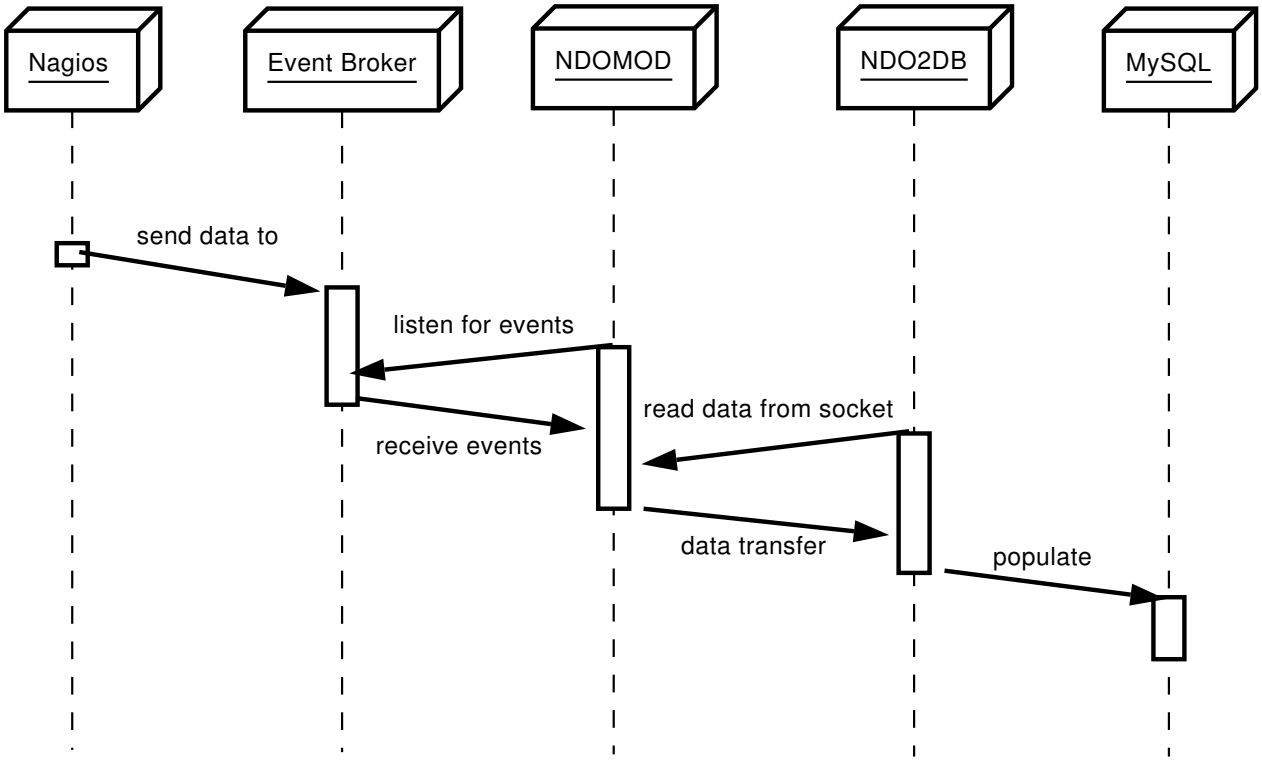

Figure 7: Populate a database with Nagios Events

a small but efficient database we have chosen SQLite as database system. Secondly, the iPhone supports the SQLite engine, so we can easily access the data within the application by using a SQLite database on the phone. Since the SQLite database should be stored on the phone we need a interface to the database server, which we will describe next.

As mentioned before the communication to the database server is problematic due to connectivity problems. Therefore we will create the small SQLite database on the server and put the file out on a web server (e.g. the Nagios web server). The application on the iPhone can now update the small database depending on the current connectivity by downloading the file. The server has no resource problem in grabbing the data from the database and transforming in a SQLite file, the iPhone would have this ressource problem. So we have shifted the information preparation in our concept on the server side.

The benefit of using the downloaded SQLite database on the iPhone lies in its independency. Even if the network connection is lost, at least the database copy is still available. So the application can use information available in the local database and count the number of tries to download the current database file from the Nagios web server. If the number of tries exceeds a certain number the user is warned that the data may not been accurate anymore. The application will still try to update the database in the background, hidden from the user.

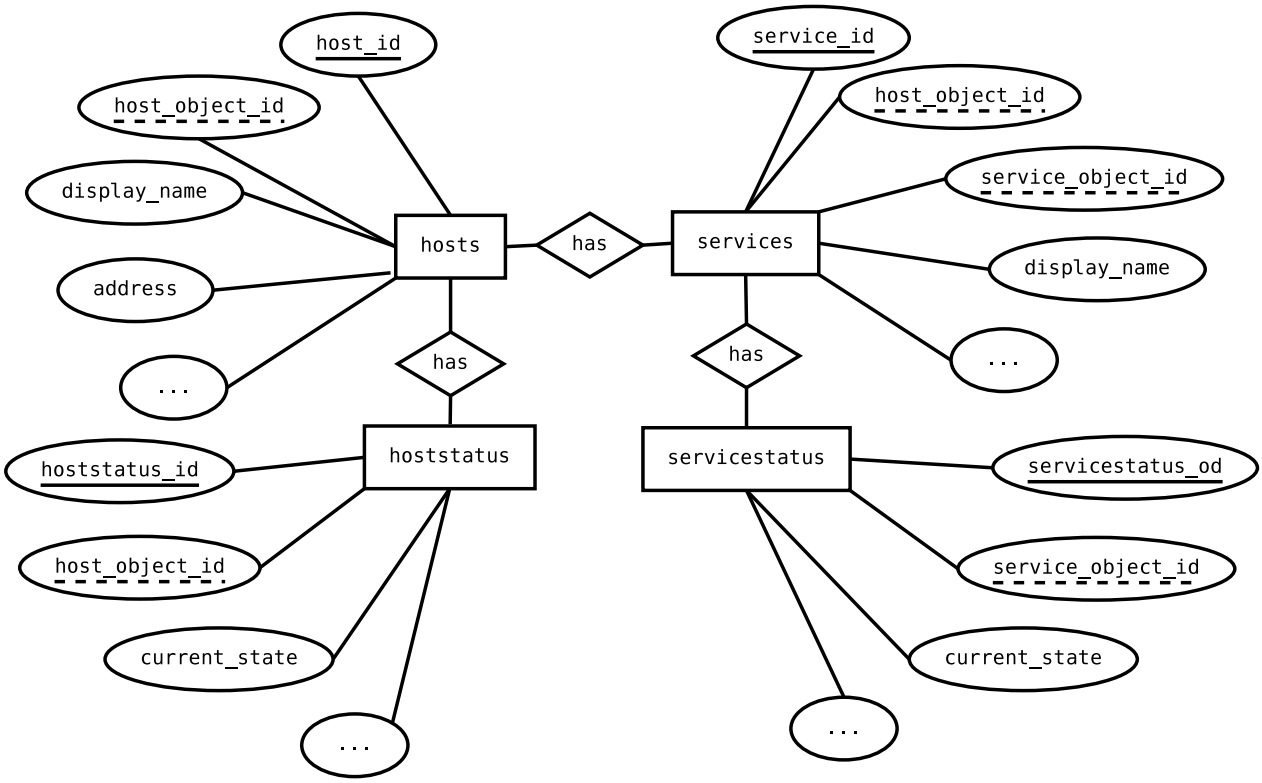

Figure 8: Database Schema of the SQLite DB on the iPhone

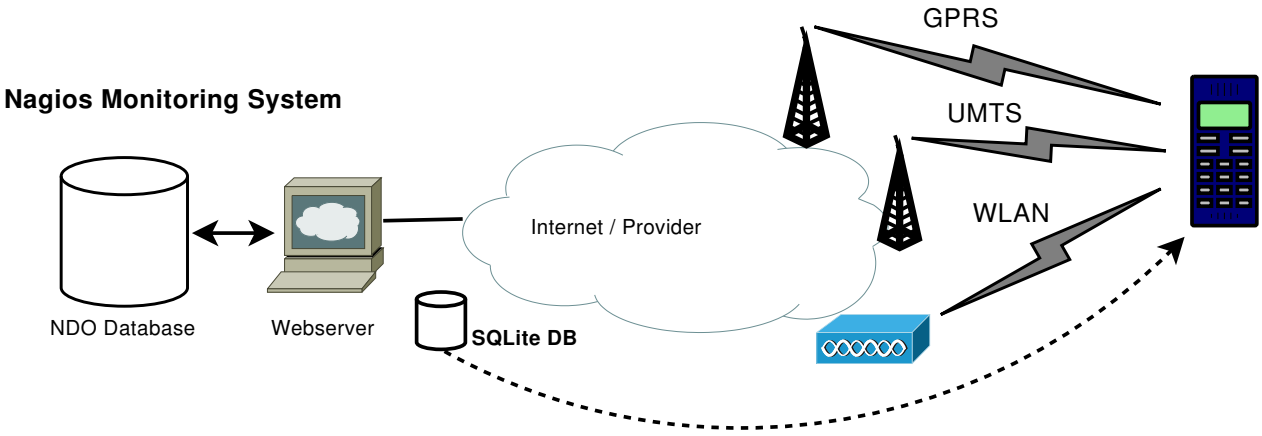

Figure 9: Information exchange between Nagios and iPhone Application

5.3.2 Application design

The application is parted in a configuration and a visualization part. The configuration part is used for giving the user an interface for changing the download address of the SQLite database on the Nagios server and for the parameters like the update-time which depends on the current network connections. The visualization part is as well parted in a host overview list (Figure 6) and the two detail visualizations host detail map (Figure 10) and service detail map (Figure 11).
The host overview list as shown in Figure 6 lists all monitored hosts by name and IP address. A green resp. red timber represents the status of the host. The user may use

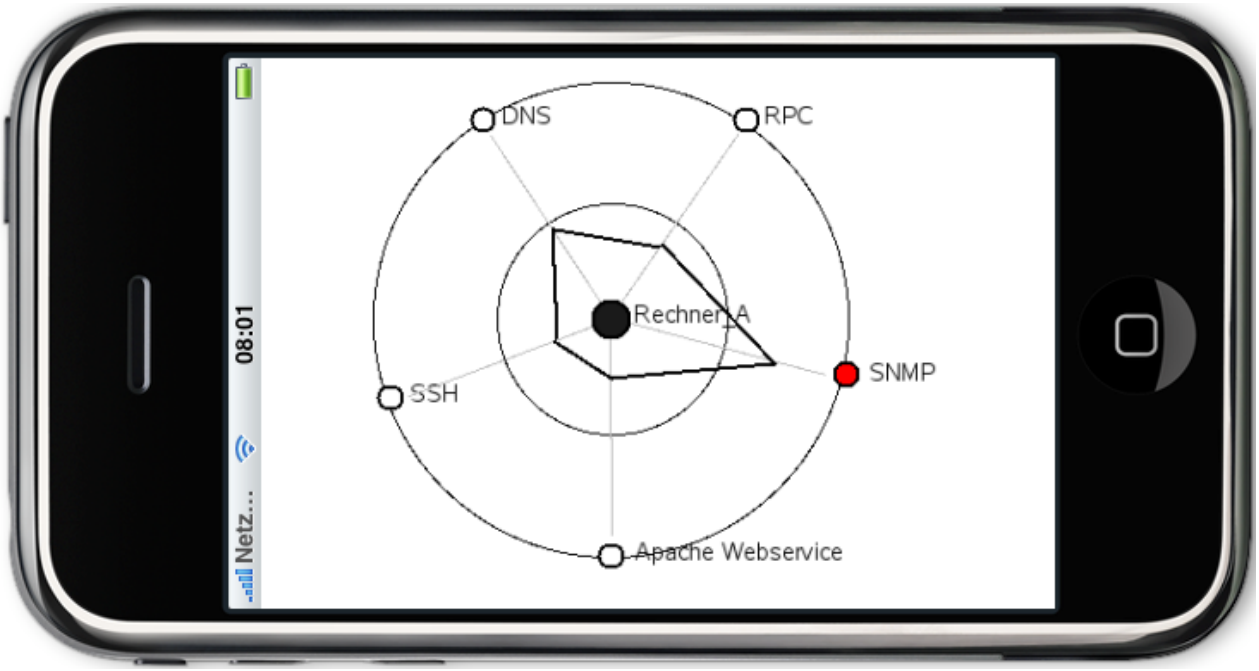

Figure 10: Shows status of monitored services on host

the arrow button on the right side of the list for switching the visualization from the host overview mode to the host detail map mode. This host detail map as shown in Figure 10 gives the user a quick overview of the status of the monitored services on the host. All services are represented by a colored circle on the outer ring of the map. The inner ring represents the target value of the service component. The point on the line between host and service circle stands for the actual monitored value. The target/actual-comparison visualized with this map, gives the user a short overview of the service status. Every value lying above the target value circle shows a service failed, making sure that the circle next to the service name is red colored.
Some services have detailed information which is visualized in a service detail map (Figure 11). In our example the SMTP Daemon is monitored and one of the supported SMTP values is above the limit. So the host detail map shows a problem for the SMTP on the host which is shown in more detail at the service status map. As seen in Figure 11 the SMTP daemon on the host monitors some value like CPU load, ..., and the free disk space. Since the amount of free space is near the total limit the target/actual-comparison shows the monitored system will soon run out of space.

5.3.3 Implementation

Currently we have implemented the host overview list and the download interface on the iPhone and the database interface on the Nagios server. The database interface consist of a little bash script, grabbing the needed information from the NDODB MySQL database and transforms the SQL-Statements in order to create the SQLite database for the application. The SQLite database file is provided by the web server on the Nagios System. The iPhone application can download the file and store it on the iPhone system as a kind of

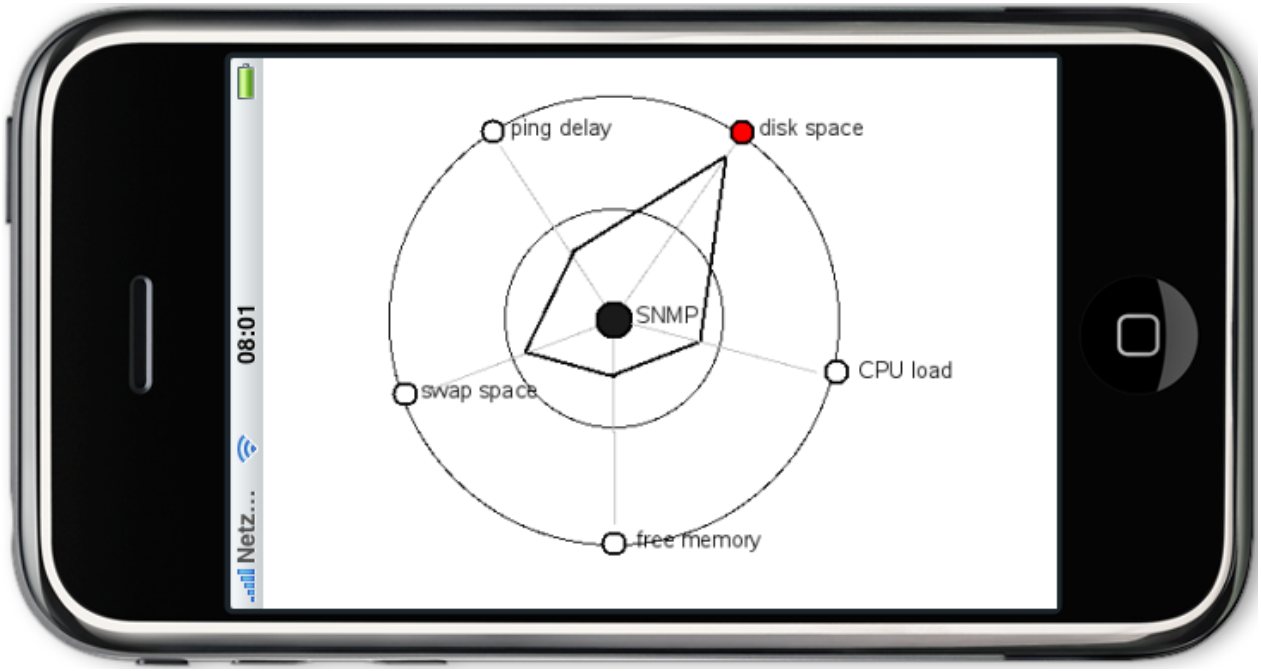

Figure 11: Shows detailed status of services

shadow database. The positive in this concept is that even if the communication is lost the user can start the application and gets at least information about the last available status of its monitored hosts and services.
The iNagMon application itself is an Apple XCode project written in Objective-C. Currently we use only the basic XCode routines as these are fitting our requirements. Next to the implemented host overview list we are currently implementing the detail maps. As these maps are using graphically visualization technique we have to cope with limiting aspects like the screen space. Concerning the available XCode libraries there is no library for painting service graphs like in Figure 10. So we have to write our own routines first to realize the detail map graphs.

5.4 Availability details

iNagMon is available from sourceforge under the GPL license[1] [7]: `http://inagmon.sourceforge.net/`

The program is split in a server and a client application. On the server side you need a web server (like Apache or httpd) and the sqlite3 program. The bash-script ”mysql2sqlite.sh” has to be executed on a periodic time-base (e.g. cron service), you may have to modify the script (e.g. path to your web server home, or mysql call to the NDO MySQL Database).

On the client side, the iPhone application is used. In the regular configuration section, you may enter the correct URI to your web server, which holds the sqlite database the server scripts periodically creates.

[1] GNU General Public License

5.5 Conclusions

iNagMon is an open-source iPhone OSX application that provides comfortable access to Nagios network monitoring data. To ensure availability of important data while on the way, we chose to build up a shadow database of the monitored data on the iPhone. We support the visualization of monitoring data in different ways like the host status list on the start screen and detailed maps.

Our data interface concept is expandable by creating update files on the server beside of an initial database file. The update files than only consist of the update SQL commands of events of a certain time period. The files should have a serial-number so that the application knows which event information to download next. Also, in a further version we are going to address security aspects by using data encryption to secure the download of sensitive data about the network.

References

[1] A. Clemm, *Network Management Fundamentals*, Cisco Press, 2006.

[2] V. Formoso, F. Cacheda, V. Carneiro, J. Valino, *Open Source Tool for Management Network Information*, Latin American Network Operations and Management Symposium (LANOMS 2007), pp.50-56, 2007.

[3] J. P. Germano, A. R. Silva, F. M. Silva, *Blackbird Monitoring System - Performance Analysis and Monitoring in Information Systems*, Proceedings of the Fourth International Conference on Web Information Systems and Technologies (WEBIST 2008), pp. 46-53, 2008.

[4] Nagios official website, http://www.nagios.org

[5] NagVis Documentation, http://docs.nagvis.org/1.3/en_US/index.html

[6] iNagios, https://inagios.com/

[7] GNU General Public License, http://www.gnu.org/licenses/gpl.html

6 SBLOMARS

6.1 Presentation

SBLOMARS consists of two software packages, namely SBLOMARS and BLOMERS. SBLOMARS is a pure decentralized monitoring system in charge of permanently capturing computational resource performance based on autonomous distributed agents. It integrates SNMP technology and thus, offers an alternative solution to handle heterogeneous resources. It also implements complex dynamic software structures, which are used to monitor from simple personal computers to robust multiprocessor systems or clusters with even multiple hard disks and storage partitions. Another feature of sblomars is that it distributes the monitoring activities into a set of sub-monitoring instances which are specific for each kind of computational resource to monitor (processor, memory, software, network and storage)

BLOMERS (Balanced Load Multi-Constrained Resource Scheduling) is a sub-optimal solution to the problem of scheduling resources for distributed systems based on a genetic algorithm. The genetic algorithm has to deal with several conditions. Basically, it has to select a set of candidate's resources from a poll, keeping individual resources performance comparative equal in all nodes of the distributed system. This restriction is intended to satisfy load balancing among the affected computational resources. In addition, the resources selection algorithm needs to keep the relative operations' sequences.

6.2 Design and Implementation

6.2.1 SBLOMARS: SNMP-based balanced load monitoring monitoring agents for resource scheduling

.

The distributed monitoring agents composing the SBLOMARS approach are pieces of software that act for a user or other program in a relationship of agency. These monitoring agents constantly capture end-to-end network and computational resources performance (processor, memory, software, network, and storage) in large-scale distributed networks. SBLOMARS is already capable of monitoring processor, storage and memory use, network activity at the interface level, services available (an updated list of applications installed), and end-to-end network traffic. This can be done across different architectures, including the Solaris, Unix-based, Microsoft-based, and even Macintosh platforms. Moreover, SBLOMERS is self-extensible and can monitor multi-processor platforms to huge storage units. Specifically, it is designed around the Simple Network Management Protocol (SNMP) to tackle the generality and heterogeneity problem, and is also based on autonomous distributed agents to facilitate scalability in large-scale Grids. SBLOMARS addresses the scalability problem by distributing the monitoring system into a set of monitoring agents specific to each kind of computational resource to monitor. Therefore, it deploys a single software thread per type of resource to be monitored, independently of the amount of such resources. This is worth to be mentioned because many monitoring

systems fail when they try to handle new, hot-plug, resources that have been added to the system. As we mentioned before, every resource is monitored by independent software threads that start again at certain intervals, becoming an infinite cycle. The cycle-timing is defined by local or remote administrators through booting parameters at the beginning of its execution.

In the SBLOMARS monitoring system, each targeted network and computational resource (memory, processor, network, storage, and applications) is considered as an autonomous shared entity, this means that a host/node forming a computational environment (Grid or Cloud) could be sharing all its computational resources (full-public), just some kind of them (partially-public), or none of them (fully-private). It is important to mention that monitoring agents are instances from the SBLOMARS monitoring system, but each kind of resource implements specific algorithms to get precise information regarding the resources usability.

The SBLOMARS monitoring system identifies the shared sources in every node of the Grid and deploys a set of monitoring agents for these specific resources. It could be better explained with an example. In real organizations, resource owners could share all of their network and computational resources or just some of them. This is possible because SBLOMARS instances a monitoring agent for each resource to share. Therefore, some workstations could offer only memory resources, and other ones could share their storage resources. This is quite common in current distributed organizations. Moreover, a user is sharing storage resources could only share part of them. This does not mean that the user is sharing all its storage capacity. For instance, in workstations it is normal to have two hard disk partitions. The operating system is running in the first one, and the users information is located in the second one. This user will share only the second partition, because he/she does not want to loose the control of the primary partition. A diagram showing the main components and interfaces of SBLOMARS monitoring agents is presented in figure 12. We now will now describe the functionality of each component and the interaction between them.

The Principal Agent Deployer is the main component of the overall monitoring system. It deploys a specific monitoring agent for each kind of resource to be monitored. It offers a generic user interface that can be used to specify the timing between every invocation of the SNMP-MIBs, as well as the number of invocations between every statistical measurement. We have named this functionality as the setting up of the polling periods. Basically, this component starts a monitoring agent for every resource to monitor. For instance, if a certain machine with two hard disks, one micro-processor, two network cards (wireless and wired), and one bank of memory will need six monitoring agents in total, then the Principal Agent Deployer will start six monitoring agents (two for the hard disks, one for the CPU, two for the network cards, and one for memory) with different polling period configurations and, obviously, each monitoring agent will retrieve different OID values from the node SNMP Agents.

The Resource Monitoring Agents component is instantiated in as many classes as there are different kinds of resources that must be tracked. We currently have six different monitoring agents. These are: memory, processor, software, storage, network interfaces (at resource level), and end-to-end network connectivity (at network level). These agents are independent from each other due to the fact that SBLOMARS can reduce the overload caused by its monitoring activity when a Grid Node is only sharing some of its resources,

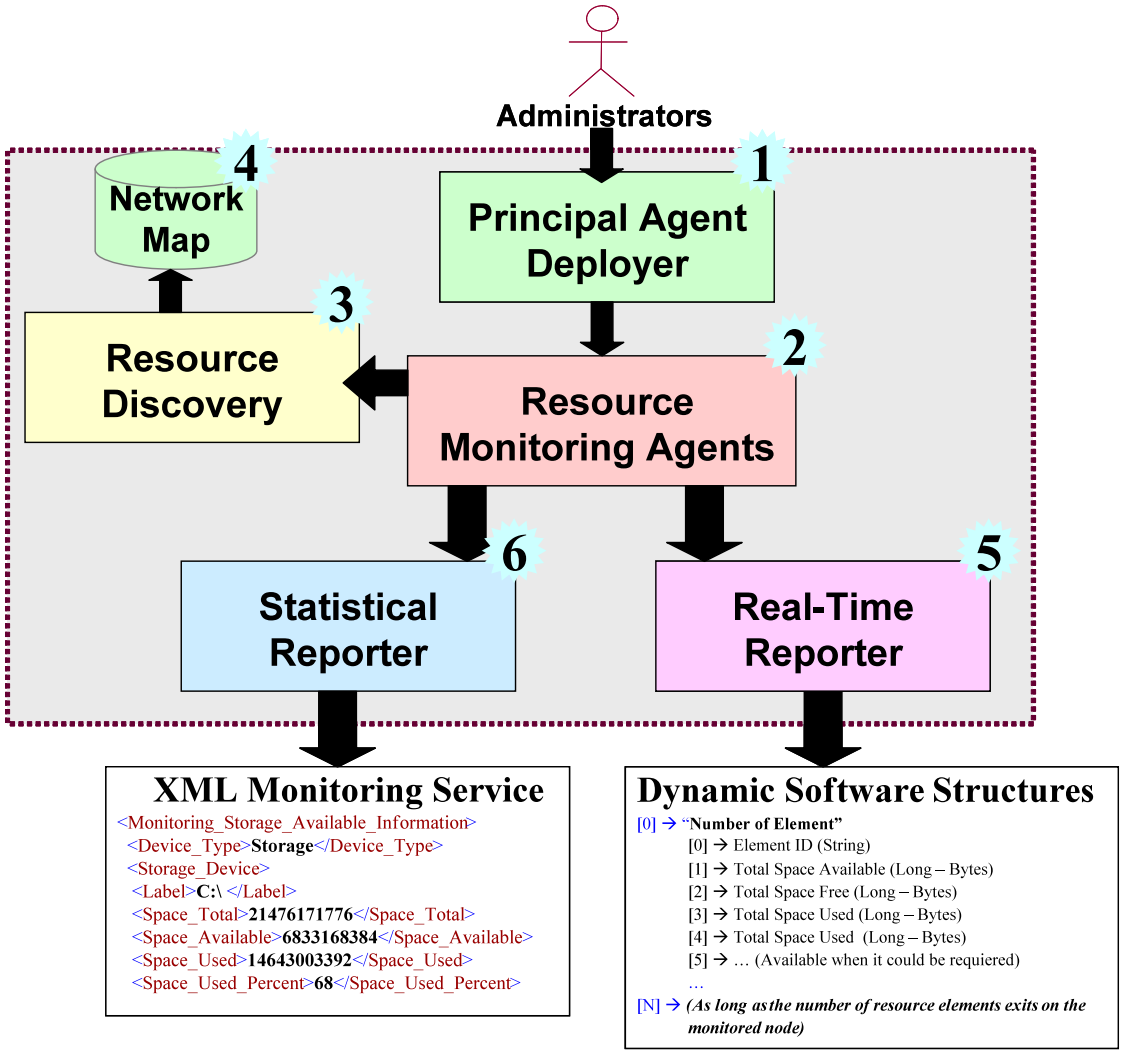

Figure 12: SBLOMARS Components and Interfaces

but not all of them. In other words, when a workstation is just sharing its storage capacity, SBLOMARS will instantiate only the sub-agent for monitoring storage devices. This ensures that SBLOMARS overload is only what is needed to monitor this resource. Other monitoring alternatives deploy as many monitoring agents as resources discovered on the workstation. Therefore, they cause an unnecessary overload on their hosting nodes.

The Resource Discovery component registers the kinds of resources available in the hosting node. We understand as hosting node, the workstation or server where SBLOMARS is being executed. It stores that information in the Network-Map Database and broadcasts its existence in order for it to be caught by higher level resource schedulers. The Real-Time Reporter component generates real-time resource availability information for each kind of resource. It publishes this information by means of Dynamic Software Structures and offers accessibility to these structures through network socket connections. These structures are available to scheduling systems (e.g. BLOMERS scheduler) or any other external instances which need to know the resources behaviour in real-time. We will thoroughly describe dynamic software structures in future sub-sections.

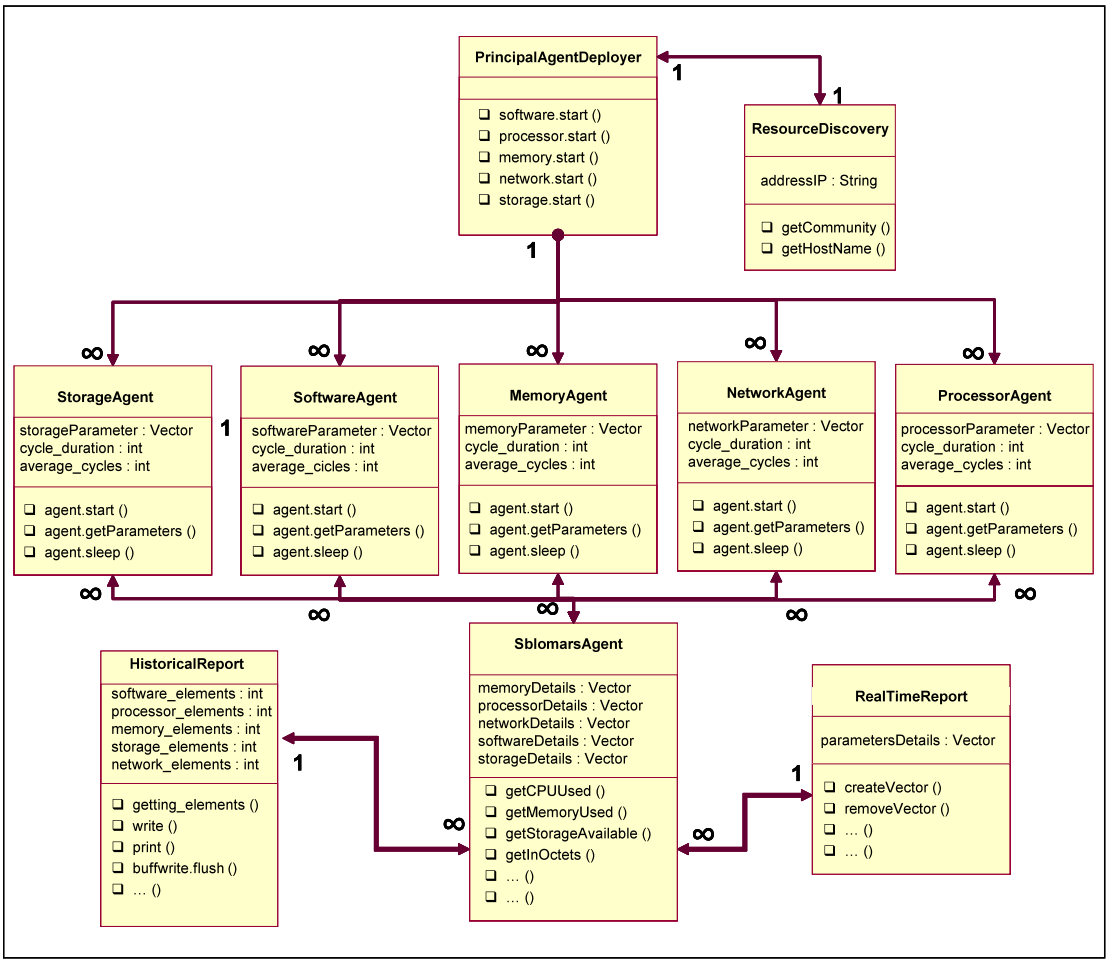

Figure 13: Diagram of Classes of the SBLOMARS Monitoring System

The Statistical Reporter component generates statistical resource availability information for each kind of resource. It publishes this information by means of the XML-based Monitoring Reports, and offers accessibility to these structures through network sockets connections. The statistical reports are later used in the resource selection phase to determine, in advance and by means of a heuristic approach, which resources are more likely to be the optimal solution for the fulfilment of any users request.

SBLOMARS has been implemented in Java. In Figure 13 we have depicted a class diagram of the designed core architecture. We have exploited the object-oriented functionality that this programming language offers to deploy a single thread per type of resource to be monitored, independent of the amount of such resources. This is worth mentioning because many monitoring systems fail when they try to handle new, hot-plug, resources that have been added to the system. As we mentioned before, every resource is monitored by independent software threads that start again at certain intervals, becoming an infinite cycle. The polling period is defined by local or remote administrators through the parameters of the Principal Agent Deployer component.

6.2.2 SBLOMARS Interfaces with SNMP-MIBs

Our monitoring agents will retrieve the required information by contacting specific objects of the available MIBs. We have used HOST-RESOURCES-MIB, UC-DAVIS-MIB, INFORMANT-MIB and CISCO-RTTMON-MIB, which are standard and well-known MIBs for networking and computing resources. The first two mentioned MIBs are configured by default when the SNMP has been installed in the system. The other two are private MIBs, but with open free access. This means than it is not necessary to pay to use them, but before using SBLOMARS it is necessary to confirm that these MIBs are already installed in the system. We then ensure that any platform will be monitored by our approach, once it has implemented the above mentioned MIBs. SBLOMARS also uses alternative open standard MIBs in order to offer full computational resource availability, but it is out of the scope of this paper to mention all of them.

6.2.3 Distributed Monitoring Agents Data Interfaces

As shown in Figure 1, the SBLOMARS outputs are presented in two formats: XML-based Monitoring Reports containing statistical resource availability information, and Dynamic Software Structures containing real-time resource availability information.

The first format, XML-based Monitoring Reports, is designed to be compatible with external systems such as resource schedulers, systems for information forecasting, and resource analyzers. In our general approach, SBLOMARS statistical resource availability reports are used for our heuristic resource scheduler system to determinate in advance which resources have a higher probability of being optimally selected for any users request.

The statistical reports are developed by means of the XML standard. Figure 14 shows an example corresponding to storage capacity in a workstation with two hard disks, but the second one has two partitions. SBLOMARS instantiates monitoring agents for each device, so, in this case, SBLOMARS has instantiated three monitoring agents.

In order to generate these reports, the Statistical Reporter component is called by the corresponding monitoring agent to translate the collected data to standard formats. SBLOMARS monitoring agents are in charge of sending the resource availability information from the last polling period to the Statistical Reporter, which collects this information until a previously specified (through Principal Agent Deployer component) number of values are collected, then the Statistical Reporter calculates the average of all the values and generates the XML-based report.

The XML-based Monitoring Reports (Figure 14) have a great advantage in terms of compatibility, but they come at a great computational cost. The activity of creating XML-based files by means of whatever programming language requires considerable CPU usage. Moreover, parsing the content of the information (parsing is the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given grammar structure) is normally quite slow and also computationally expensive.

SBLOMARS monitoring agents include an alternative format for interface resource behaviour information. We have designed a set of dynamic structures which are self-extensible

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Edited with Agent SBLOMARSXML v1.0 …
<!-- Monitoring Resources Service xmlns:xsi= …
<Monitoring_Storage_Available_Information>
  <Device_Type>Storage</Device_Type>
  <Number_of_Elements>3</Number_of_Elements>
   <Storage_Device>
    <Label>C:\ Label: Serial Number f010b634</Label>
    <Space_Total>21476171776</Space_Total>
    <Space_Available>6833168384</Space_Available>
    <Space_Used>14643003392</Space_Used>
    <Space_Used_Percent>68</Space_Used_Percent>
    </Storage_Device>
  <Storage_Device>
    <Label>G:\ Label:Disco local Serial Number 302e</Label>
    <Space_Total>10733957120</Space_Total>
    <Space_Available>3095842816</Space_Available>
    <Space_Used>7638114304</Space_Used>
    <Space_Used_Percent>71</Space_Used_Percent>
    </Storage_Device>
  <Storage_Device>
    <Label>H:\ Label:SHARED Serial Number 48f893</Label>
    <Space_Total>34290843648</Space_Total>
    <Space_Available>13172244480</Space_Available>
    <Space_Used>21118599168</Space_Used>
    <Space_Used_Percent>61</Space_Used_Percent>
  </Storage_Device>
</Monitoring_Storage_Available_Information>
```

Figure 14: XML-based Report of Storage Availability Information

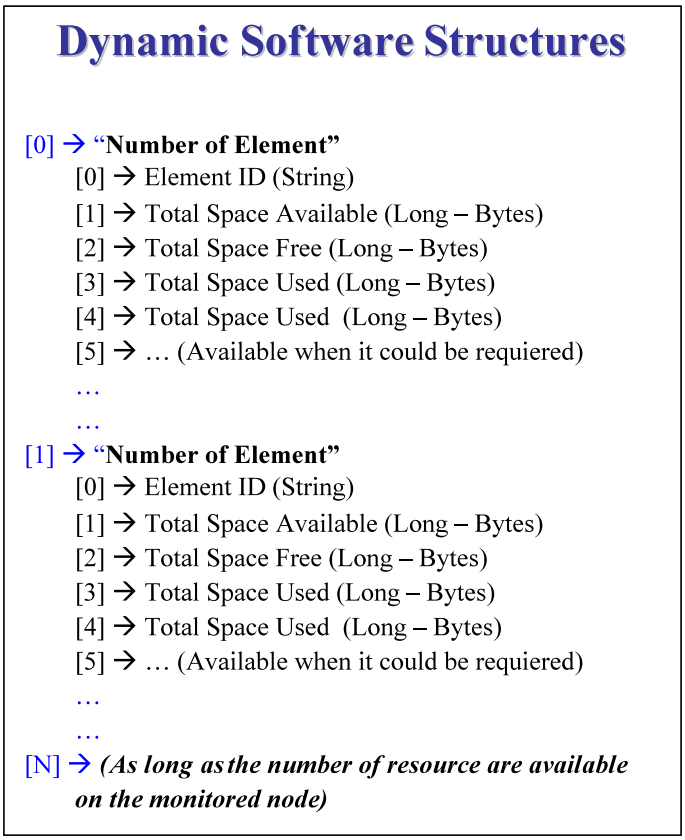

Figure 15: Dynamic Software Structure Example

regardless the number of new components they involve. We have named this solution Dynamic Software Structures. They contain real-time resource availability information. In order to facilitate the scalability of the SBLOMARS monitoring agents, these structures are able to add system components as needed, reducing the load where necessary. The Dynamic Software Structures (Figure 15) are software structures that keep in the memory buffer real-time resource behaviour information from the last polling period. This information remains available until the next resource availability information request. This is another advantage of this approach; any external instance could get this information from the memory buffer in a faster way than accessing the XML-based reports directly. As we have mentioned before, the polling period is assigned by the local or remote node administrator (resource owners) at the moment of configuring the monitoring agents in the Principal Agent Deployer component.

6.2.4 Distributed Monitoring Agents and CISCO IP SLA Integration

SBLOMARS offers end-to-end jitter, packet loss rate, bandwidth, and delay average between a Cisco Switch/Router and any IP point. SBLOMARS could also be configured to monitor end-to-end links with different classes of service (DiffServices). Basically, Cisco IOS IP SLA uses active monitoring, which includes the generation of traffic in a continuous, reliable, and predictable manner. Cisco IOS IP SLAs actively send data across the

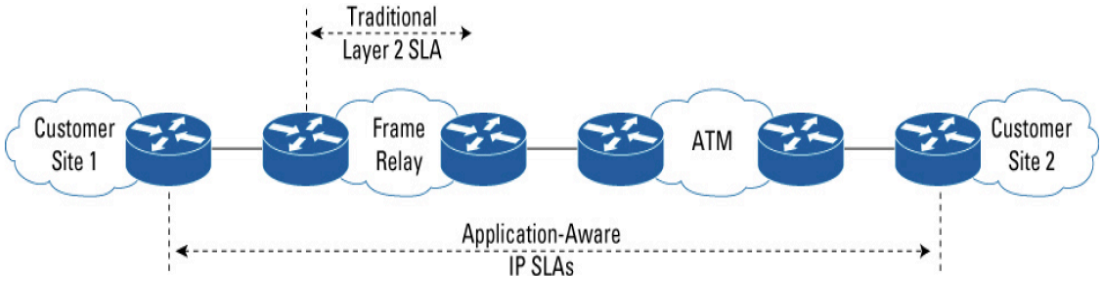

Figure 16: Traditional SLAs versus Cisco IOS IP SLAs

network to measure performance between multiple network locations or across multiple network paths, as we show in Figure 16. It uses the timestamp information to calculate performance metrics such as jitter, latency, network, and server response times, packet loss, and voice quality scores.

The user defines an IP SLAs operation (probe) within Cisco IOS (this interface has been developed as a part of this research) to create and deploy these probes. In this interface we have included measurement characteristics such as packet size, packet spacing, protocol type, Diff-Serv Code Point (DSCP) marking, and other parameters. The operations are scheduled to generate traffic and retrieve performance measurements. The data from the Cisco IOS IP SLAs operation is stored within the RTTMON MIB and is available for Network Management System applications to retrieve network performance statistics. Cisco IOS IP SLAs is configured to monitor per-class traffic over the same link by setting the DSCP bits. A destination router running Cisco IOS Software is configured as a Cisco IOS IP SLAs Responder, which processes measurement packets and provides detailed timestamp information. The responder can send information about the destination routers processing delay back to the source Cisco router. Unidirectional measurements are also possible using Cisco IOS IP SLAs.

Cisco IOS IP SLAs provides a proactive notification feature with an SNMP trap. Each measurement operation can monitor against a pre-set performance threshold. Cisco IOS IP SLAs generates an SNMP trap to alert management applications if this threshold is crossed. Several SNMP traps are available: round-trip time, average jitter, one-way latency, jitter, packet loss, and connectivity tests. Administrators can also configure Cisco IOS IP SLAs to run a new operation automatically when the threshold is crossed. When a set of probes have been deployed through our SNMP-GUI, CISCO IOS IP SLAs starts to collect metrics regarding network performance and, as we mentioned before, it stores them in the CISCO-RTTMON-MIB. All these values are not functional network monitoring information yet. There are some interpretations and calculations that SBLOMARS has to make in order to get proper networking performance metrics. Our monitoring agents retrieve the fundamental values to get the jitter, delay, packet loss,and bandwidth. We show an example to get Average Jitter. (DS is destination to source, SD is source to destination and the result is given in milliseconds) in the segment of code in Figure 17.

```
public int getJitter(String [] oIDValRTTMon) {
try {

jitterSum = sumOfPositiveDS +
   sumOfNegativeDS +
   sumOfPositiveSD +
   sumOfNegativeSD;

jitterNum = numOfPositiveDS +
   numOfNegativeDS +
   numOfPositiveSD +
   numOfNegativeSD;

avgJitter = jitterSum/jitterNum;
}
}
```

Figure 17: Pseudo Code to Calculate Average Jitter by SBLOMARS

6.3 BLOMERS: Balanced Load Multi-Constrain Resource Scheduling System

The BLOMERS scheduling system deals with several conditions. Basically, it selects a set of candidate resources from a poll, keeping individual resource performance comparatively balanced in all nodes of the cluster or Grid. This condition has been added in order to satisfy computational resource load balancing. In the BLOMERS resource scheduler approach, we propose a sub-optimal solution to the problem of scheduling computational resources in large-scale computing environments, namely a resource scheduler based on a Genetic Algorithm (GA) implementation.

6.3.1 The BLOMERS Genetic Algorithm Design

Genetic Algorithms are a very simple methodology to solve NP-hard problems. The flowchart for our algorithm design is shown in Figure 18. Basically, there are two fundamental activities in this flowchart. The first one is the Valuation of the generated population and the second one is reproduction into a new population.

The pseudo code of the heuristic resource scheduling algorithm is depicted in Figure 19.

BLOMERS uses a collection of solutions (population) from which better solutions are produced using selective breeding and recombination strategies. The first population is created randomly by means of the Initialize (k, Pk) method. Where k is the kind of resource to analyze and Pk is the population of k resources selected.

The method Initialize (k, Pk) calls a random method to select the first population of our solution. As BLOMERS is implemented in JAVA code, we have used the primitive random function in this method. Once the first population has been initialized, a first simple

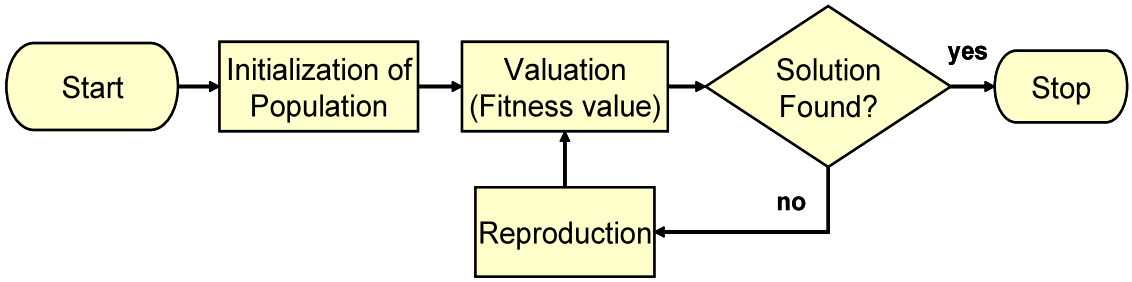

Figure 18: The Flowchart of BLOMERS Genetic Algorithm Design

```
CleaningBuffer (Pk)
Initialize (k, Pk);
Evaluate (Pk);
     Do
       {
       Select_Resource_Candidates (Pk);
       Crossover (Pk);
       IF Evaluate (Pk+1) == Minimal Constraints;
          Ends Do-While;
       ELSE
          Mutation (Pk);
       IF Evaluate (Pk) == Minimal Constraints;
          Ends Do-While;
          }
Deliver (k_solution);
```

Figure 19: BLOMERS Genetic Algorithm Pseudo Code

STORAGE		PROCESSOR
ID_0: 147.83.106.199:6400		ID_0: 147.83.106.199:6000
ID_1: 147.83.106.199:6401		ID_1: 147.83.106.201:6000
ID_2: 147.83.106.201:6400		...
...		
MEMORY		**NETWORK INTERFACES**
ID_0: 147.83.106.199:6200		ID_0: 147.83.106.199:6800
ID_1: 147.83.106.201:6200		ID_1: 147.83.106.201:6800
...		...

Figure 20: List of Shared Resource from Workstation A and Workstation B

STORAGE		CHROMOSOME
ID_0: 147.83.106.199:6400		0000 0000 00
ID_1: 147.83.106.199:6401		0000 0000 01
ID_2: 147.83.106.201:6400		0000 0000 02
...		...

Figure 21: Encoding Example of Storage Resource from Workstations A and B

evaluation of this population is done. Normally, the first population is never selected as a candidate solution, but it is the main entry to create the new populations. The Select Resource Candidates (Pk) method bounds the initial populations and applies two simple genetic operators, such as Crossover (Pk) and Mutation (Pk). These methods are used to construct new solutions from pieces of the old one in such a way that the population (Pk) steadily improves.

6.3.2 Encoding Solution of the BLOMERS Genetic Algorithm

Figure 20 is an example of a representation of the network map considering that there are only two workstations. Actually, in this table we show four files because BLOMERS scheduling system generates one network map file per kind of resource that is discovered in collaboration with SBLOMARS monitoring system.

The encoding of resource availability information is based on Figure 9. In our approach we substitute these referenced IDs (0, 1, 2, , N) by a 10-bits binary code. The binary code used is just the binary representation of the ASCII number which represents the IP address and the port of the resource available within the Grid Infrastructure. In order to maintain the scalability of the scheduler, we have implemented a dynamic coding system. Therefore, each Chromosome represents a resource, using strings of 0s and 1s. The Population is the number of Chromosomes available to test. In Figure 21 we illustrate a coding example from the STORAGE network map file from Figure 20.

			CHROMOSOME
ID: 24	Parent 1	=	0000 0110 00
ID: 49	Parent 2	=	0000 1100 01
…	…		…

Figure 22: Example of the Random Selection of Two Chromosomes Population

			CHROMOSOME
Storage_0 → ID_24: 147.83.106.199:6400	Parent 1	=	0000 0110 00
Storage_1 → ID_45: 147.83.106.167:6401	Parent 2	=	0000 1100 01

Figure 23: Example of Population Evolution Before Crossover Operation

6.3.3 Crossover Operation

Here, selected individuals are randomly paired up for crossover (sexual reproduction). This is further controlled by the specified crossover rate and may result in a new offspring individual that contains genes common to both parents. New individuals are injected into the current population. Crossover operators in our approach exchange substrings of two parents to obtain two offspring. In the BLOMERS scheduling system, the crossover operator combines useful parental information to form new and hopefully better performing offspring. Such an operator can be implemented by choosing a point at random, called the crossover point, and exchanging the segments to the right of this point. For example, let be the two randomly selected parents 24 and 49, which are associated to Storage 0 and 1 respectively as shown in Figure 23.

The resulting offspring would be:

In the example shown in Figures 23 and 24, Child 1 and Child 2 will be the resulting population. The advantage of using the proposed crossover operation is that we can select individuals from the same Virtual Organization. Here, the selection mechanism takes into account neighbor nodes when they are not processing high amounts of tasks.

6.3.4 Mutation operations

In this step, each individual is given the chance to mutate based on the mutation probability specified. If an individual is to mutate, each of its genes is given the chance to

			CHROMOSOME
Storage_1 → ID_25: 147.83.106.199:6401	Child 1	=	0000 0110 **01**
Storage_0 → ID_44: 147.83.106.167:6400	Child 2	=	0000 1100 **00**

Figure 24: Example of Population Evolution After Crossover Operation

			CHROMOSOME
Storage_1 → ID_25: 147.83.106.199:6401	Parent 1	=	0000 0110 01
			MUTATION
Storage_N → ID_57: 147.83.206.199:6401	Child 1	=	0000 1110 01

Figure 25: Example of Population Evolution with Mutation Operation

randomly switch its value to some other state. In BLOMERS resource scheduler algorithm the mutation operator randomly alters each Chromosome with a small probability, typically less than 1%. This operator introduces innovation into the population and helps prevent premature convergence on a local maximum. The evolution is terminated when the population attains certain criteria, such as simulation time, number of generations, or when a certain percentage of the population shares the same function value. In our approach a clear example is as follows:

In the example of Figure 25, Child 1 will be the resulting population. The difference from the previous operation is that the mutation operation helps to move from one Virtual Organization or Administrative Domain in the Grid towards others with similar resource availability information.

6.3.5 Selection Mechanism

Here the performance of all the individuals is evaluated based on the fitness function, and each is given a specific fitness value. Here it is important to mention that the higher the value, the bigger the chance of an individual passing its genes to future generations. Through the overview section of this Chapter, we have mentioned that Genetic Algorithm could apply several selection methods. In BLOMERS scheduling system we have chosen the Elitism selection, in order to distribute jobs as fairly as possible.

In the proposed and implemented Elitism selection method, we have chosen a group of individuals from a population. This group is composed of seven individuals that are selected at random from the population, and the best (normally only one, but possibly more) is chosen. The best is the one that uses, at the moment of the scheduling, the least amount of its shared resources. This methodology is considered an exhaustive search. In this case, the exhaustive methodology for scheduling does not take time that grow exponentially because the number of resources to compare is restricted to the offspring population. This population, as we said before, is composed of only seven Chromosomes (candidate resources).

Fitness is a measure of how well an algorithm has learned to predict the outputs from the inputs. The goal of the fitness evaluation is to give feedback to the learning algorithm regarding which individuals should have a higher probability of being allowed to multiply and reproduce, and which of them should have a higher probability of being removed from the population.

This algorithm is faster than other heuristic methodologies, and could be better adapted to heterogeneous parameters, but the most important advantages are that it avoids failing

into a local minima solution, and it can be run in parallel. We made use of this advantage to design our genetic algorithm in many threads as many resource devices have been found on the VO.

Our approach is not as effective as exhaustive search approaches, but it is faster and lighter in terms of computing performance impact. Basically, every genetic algorithm has a selection probability (Ps) function based on its fitness value for each solution. The coefficient between the fitness value of a specific solution and the summary of all the fitness values of the same one is the selection probability for this solution:

$$Ps(i) = \frac{f(i)}{\sum_{i=1}^{N} f(i)} \quad (1)$$

Applying this methodology is not a simple task. The genetic algorithm needs to be adapted according to the requirements of the application and the environment in which it will be working. The information to analyze for each search will be adaptable to resource availability and the conditions for this adaptation are completely different in each design, which ensures the novelty and originality of this approach.

6.4 Availability details

SBLOMERS is documented and publicly available under a GPL license at `http://nmg.upc.edu/sblomers`.

6.5 Conclusions

SBLOMARS is stronger than typical distributed monitoring systems in three essential areas: First, it reaches a high level of generality via the integration of SNMP technology, and thus, we are offering an alternative solution to handle heterogeneous operating platforms. Second, it solves the flexibility problem by implementation of complex dynamic software structures, which are used to monitor simple personal computers to robust multiprocessor systems or clusters, even with multiple hard disks and storage partitions. Finally, the scalability problem is covered by the distribution of the monitoring system into a set of monitoring agents, which are specific to each kind of computational resource to monitor. The implemented resource scheduling system based on Genetic Algorithms BLOMERS is able to search for a sub-optimal set of resources for any requested Grid Service, starting the matching process from a set of resources, rather than from just one at a time. This parallelism has been introduced to avoid the Genetic Algorithm be trapped on local minima, which means that the scheduling system will be searching and matching the whole set of shared resources from the Grid Infrastructure at any given time. BLOMERS is highly customizable. In our algorithm parameters like thresholds, end-to-end network performance, statistical resource availability information, and software available are controlled by the administrator of the scheduling system.

7 Lambda Monitor packaging and distribution

7.1 Presentation

The Lambda Monitor (LM) software is wavelengths monitoring solution which makes it possible to supervise many wavelength circuits simultaneously and to notify (real-time) about current problems occurring in the optical networks. Using the Lambda Monitor graphical web-based interface, an operator can analyze in a quick and exact manner which part of the lambda circuit (exactly which device) is causing problems. The web interface also provides the user with additional information about the optical characteristics of individual wavelengths as well as information about the optical device.

The LM web interface consists of the three levels (called layers), these are the Lambdas layer, the Devices layer and the Parameters layer. The first part of the user interface called ”Lambdas layer” shows the current status of the monitored wavelengths circuits (end-to-end). The Lambdas layer provides the user with an integrated view of all monitored wavelength circuits. The second level of the user interface is the ”Devices layer” where the lambdas are presented in the device-oriented view. The interface's level shows the current status of the DWDM devices' interfaces terminating selected wavelength. The ”Parameters layer”, the most detailed level of user interface presents the current values of various monitored parameters on selected device's interface in the form of graphical tables. The tables consist of detailed information about many parameters, which determine and influence the operation of lambdas. Parameters layer enables users also to reach the archived values of the 15Min and 24h intervals parameters. Lambda Monitor has the ”Current alarms” module as well, which enables viewing information about alarms that occur currently in the monitored lambda circuits. The ”Alarms history” module presents all the alarms that occurred formerly. The main advantage of the visual representation of each wavelength is that it provides a simple visual indication to the viewer of the current status of a wavelength without having to understand large numbers of complex alarm messages from numerous DWDM network elements and line cards. Hence LM is excellent at hiding the complexity of the network from the viewer.

This software is currently deployed in European NRENs PIONIER (Poland) and HEAnet (Ireland) as well as used in GEANT2 network. It was successfully presented at international conferences (INGRID2007 and AICT2008).

Lambda Monitor functionality has been extended under EMANICS project WP6 call2 as shortly described below: - Parameters archive - archiving of current values of various monitored parameters and visual display of gathered data in the form of graphical charts. - Thresholds monitoring - monitoring and notification of configured thresholds exceedance. - Mail notification - notification of current alarms occurring in monitored wavelengths by e-mail. - Full documentation of the system.

Aforementioned funtionality is currently fully operational and accessible under the Lambda Monitor project homepage: `http://lmonitor.man.poznan.pl`

7.2 Expected Impact

The core application has been partially developed with the support of EMANICS WP6 and packaging process made Lambda Monitor easily available for broad public enabling more deployment within different networks. It was a very important step in the development process, as it allows wider group of users to download, setup and test this tool. We expect that with a still growing community we will be able to achieve subsequent milestones faster and with a better quality.

The number of useful tools that are dedicated for optical network management and monitoring has been increased by adding the tool to EMANICS software repository and inventory. The description of the LM software is available in EMANICS repository at `http://emanics.org/index.php?option=com_dbquery&Itemid=93&task=ExecuteQuery&qid=3&previousQu1`

Moreover the NoE visibility has increased thanks to having its name mentioned in the LM documentation.

7.3 Progress Report

Thanks to the support of EMANICS the source code was reviewed, corrected and commented, to help prospective developers and other interested people read through it. The documentation of the project was changed to reflect the present state of the code and now describes available configuration options.

Besides providing the source code in tarball archive, the packages for specific Unix-based operating systems were created in order to ease the installation and deployment process on these platform. Currently a package in DEB format for Debian-based distributions (Debian, Ubuntu, Kubuntu etc.) as well as RPM format RedHat-based distributions (RedHat, CentOS, Fedora) exist.

7.4 Conclusion

Most of the available optical network monitoring solutions are proprietary and single vendor solutions only designed for network operators. In our opinion neither of them provide a clear and graphical end-to-end monitoring of individual wavelengths.They are also often limited in functionality to one-domain homogeneous network environment only.

The Lambda Monitor is based on the GNU GPL version 3 license. The open source license is one of the main features that distinguish Lambda Monitor from other currently available wavelengths monitoring tools.

8 LINUBIA: Linux User-Based IP Accounting

Obtaining information about the usage of network ressources by individual users forms the basis for establishing network billing systems or network management operations. While there are already widely used accounting techniques available for measuring IP network traffic on a per-host basis, there is no adequate solution for accounting per-user network activities on a multiuser operating system. This work identifies requirements for a user-based IP accounting module and develops a prototypical implementation for the Linux 2.6 operating system, which is capable of providing per-user accounting for both the IPv4 and the IPv6 protocol.

This section provides a brief presentation of LINUBIA. A more detailed description can be found in [1].

8.1 Expected Impact

Several use cases have been identified where LINUBIA can provide support in the accounting process. As it provides a user-based IP accounting with a very low impact on system and network performance, LINUBIA can be seen as an appropriate choice for any situation that maps to one of the scenarios below.

8.1.1 Network Traffic Billing System

The first scenario deals with the case of a grid infrastructure spanning across a larger area on top of which customers may run their own grid applications. A grid user will typically install its applications on multiple nodes and these run typically with the users privileges. The grid operator may use the user-based accounting module in order to split network costs (traffic created by grid applications is typically high) among all customers based on the amount of traffic they created.

8.1.2 Individual Load Monitoring and Abuse Detection

The second scenario addresses the case of an institution, for example a university, which offers its students the possibility to use the Web for research and communication purposes, but does not want them to excessively waste precious network bandwidth for sharing videos, filesharing, and the like. The system setup is done in a way that a student can log into one of many computers at the university with his personal credentials. The user account information is stored in a centralized LDAP directory, so a specific student has uses the same user ID (UID) in every system he logs into. A script can regularly copy usage information to a database server, where it is stored and accumulated with the traffic footprint of other users in order to detect possible anomalies in the traffic under investigation. The system administrator has the possibility to monitor network usage of students, independent of applications or the computer they use. With the help of this information he can detect and quantify abuses, suspend accounts of the respective users, or initiate further investigations.

8.1.3 Service Load Measurement

The third scenario handles the identification of applications, which generate abnormal traffic. For example, on a Linux server different services may be operational, some of them may not be using well-known ports (e.g., a bit-torrent client, which constantly changes ports it is running on). On that router connecting this server to the Internet, the administrator can monitor how much traffic this server created, but he can only identify applications based on port numbers. In case of applications that change these ports the use a user-based IP accounting module eases traffic monitoring for these type of applications.

8.2 Debian Package

LINUBIA functionality is embedded in the LINUX kernel. A debian package has been prepared that contains the Linux 2.6.17 kernel source including LINUBIA as well as package containing a compilation of the respective kernel with user based ip accounting enabled for IPv4 and IPv6.

The two packages are called:

```
kernel-source-2.6.17-linubia_1.0_i386.deb
kernel-image-2.6.17-linubia_1.0_i386.deb
```

The two packages can be downloaded from http://www.csg.uzh.ch/staff/morariu/linubia/.

In order to install the kernel image the following steps are required:

```
1) Download kernel image
wget http://www.csg.uzh.ch/staff/morariu/linubia/kernel-image-2.6.17-linubia_1.0_i386

2) Install the new kernel
dpkg -i kernel-image-2.6.17-linubia_1.0_i386.deb
```

After rebooting the information about the user-based IP traffic accounting is available through the proc filesystem using the files /proc/net/ip4_usertraffic and /proc/net/ip6_usertraffic.

LINUBIA is released under a GPL (General Public License)license.

8.3 Conclusions

LINUBIA is a prototypical implementation of an IP accounting approach for modern Linux (2.6 series) operating systems. It can be used with the IPv6 network protocol and it can be integrated into an existing accounting infrastructure, such as Diameter. The current implementation shows a clear proof of concept. Compared to traditional device-based accounting mechanisms, a user-based approach allows the mapping of network services usage not only to a device, but more specific, to the user which consumed those services. Improvements are possible, e.g., with the storage component, which can be done with a

smaller memory footprint and also more efficiently by utilizing advanced data structures that will help to optimize access times. Another interesting issue determines the linkage of the networking subsystem to the socket interface, which also implies a link to the process management of the operating system. An advanced accounting module can offer IP accounting not only per user, but also per process. This allows for the identification, the management, or schedulability of processes not only by their CPU usage or memory consumption, but also by their network resource consumption. Finally, this leads to the creation of network filters or firewalls that allow for or deny network access to specific applications or users running on a host, instead of only allowing or denying specific services. The current LINUBIA implementation treats all traffic the same, thus producing an overall network consumption report for each user. An interesting improvement would be separated accounting for different services (differentiated based on DSCP number or destination Autonomous System).

References

[1] C. Morariu, M.Feier, B. Stiller: LINUBIA: A Linux-supported User-Based IP Accounting. DSOM 2007, 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2007), San Jose, USA, October 2007.

9 Conclusions

The last call for Open Source development and/or packaging made within work-package 6 in its fourth year was very successful since it received 14 submissions for developments and 5 for packaging. This led to a strong selection with a selection rate of 35% for development and 40% for packaging. All funded activities did start immediately after the selection in january 2009. All except one initiative (LATTICE) are now completed and all except one supported software components are available to the community for download.

10 Acknowledgement

This deliverable was made possible due to the large and open help of the WP6 Partners of the EMANICS NoE.

Appendix 1: List of submitted software development proposals in call 3 of work-package 6

#	Partner	Fund Request	Description
1	UniBwM	9K	A Nagios Monitor for the Iphone. The objective is to provide access to Nagios through an Iphone by building a dedicated web-app for the Iphone
2	UniZH	8.5K	Liveshift : a P2P vido streaming platform
3	UniZH	8.5K	Porting of the FastSS Similarity Search search engine on the Android platform
4	JUB	10K	Design and implementation of a Manager side Netconf Python API
5	UPC	10K	SBLOMERS A monitoring and load balancing framework for grid management
6	INRIA	10.2K	SecSip SIP firewall support for SNMP-based interaction with device agents to check availability and status
7	INRIA	10K	SECSIP2 : development of a Netconf-compliant management interface for the SecSiP Open Source Firewall
8	JUB	10K	SNMPSyslog integration : NetSNMP support of JUB RFC SNMP messages representation in structured elements of SYSLOG.
9	JUB	10K	Extension of the Open SSH Platform to support session resumption.
10	PSNC	8.5K	Weathermap : extension of an existing internal weathermap software.
11	JUB	10K	YANG_LIBSMI : A yang parser for libSMI
12	INRIA	10K	Yang / ENSUITE support : the objective is to make the ENSUITE Netconf toolkit YANG aware by providing within the compiler mapping to ENSUITE structures from YANG spcifications and building the necessary support in the agent Framework.
13	UCL	-	A monitoring Monitoring Platform for Grid Ressources
14	UCL	-	NDQOS : Network Dimensioning and Quality of Service Optimization Service. Extension of an existing toolkit.

Appendix 3: Software packaging call 3

EMANICS Work Package 6 : Open Source Software Packaging & Tutoring

EMANICS Work Package 6 : Open Source Packaging
This call covers the last 12 months of EMANICS.
It has a overall maximum budget of 9K Euros.
Proposal Sheet

The proposal has to be filled & sent to the WP Leader : Olivier Festor, (Olivier.Festor@loria.fr) before January, 16th 12 AM. The flat funding for Packaging & Tutoring will be of 3K per initiative.

Every supported initiative commits to provide 1 months before the end of the 12 months supporting period, a detailled description of the addressed software (2-4 pages) together with a precise presentation of the packaging efforts made (2-3 pages) to be included in the deliverable of the project. This is to be provided in the WP6 deliverable format which in Phase 3 will be LATEX.

Note: fill the proposal carefully and provide detailed, precise and measureable commitments. Incomplete proposals will be immediately rejected.

==========

0. Proposal Title:

1. Addressed Software

name & short description of the existing Open Source project addressed

3. Detailed List of Packaging & Documentation Activity planed under the support of EMANICS

(describe all tasks planed and extensions envisioned as part of this support)

4. Expected Impact

(what new "markets" the enhanced software will "conquer", how many distributions are envisioned, where is it going to be integrated, what visibility the NoE can gain through this support ?

Appendix 4: Software development call 3 text

EMANICS Work Package 6 : Open Source Software Initiatives
This call covers the last 12 months of EMANICS(January-December 2009)
It has a overall maximum budget of 51K Euros.

Proposal Sheet

The proposal has to be filled & sent to the WP Leader : Olivier Festor, Olivier.Festor@loria.fr) before January 16th 12 AM. Cooperative Open Source developments will be favored over proposals including a single participant. In order to fund several proposals, collaborative requests should not exceed 15K. Single request should not exceed 8.5K/request).

Every supported initiative commits to provide 1 months before the end of the 12 months supporting period, a detailled description of the software (5-10 pages), a precise presentation of the made changes (2-3 pages), and an impact evaluation (1-2 pages) to be included in the deliverable of the project. This is to be provided in the WP6 deliverable format which in Phase 3 will be LATEX.

Note: fill the proposal carefully and provide detailed, precise and measureable commitments. Incomplete proposals will be immediately rejected.

==========
0. Proposal Title:

1. Overall Open Source Software Description

(A 1/2 to 3/4 page description of the concerned software, its current status, its visbility, its use, ... In case of a non existing soft the emphasis should be made on its need and its impact on the community)

2. Licensing & distribution scheme

(license type & distribution scheme used for the software : GPL, LGPL, QPL, ...; available on a given forge, is it or will it be embedded in a third party software distribution, ..., ...)

3. Detailed List of Extensions Planed under the support of EMANICS

(describe all tasks planed and extensions envisioned as part of this support)

4. Expected Impact

(what new "markets" the enhanced software will "conquer", how many distributions are envisioned, where is it going to be integrated, what visibility the NoE can gain through this support ?

5. Cooperation level

(which parts of the extensions planed come from a cooperation among one or more partners in EMANICS, e.g. X will integrate in his software the algorithm defined by Y).

6. Cost & Requested support

(Expected cost overall + requested support) Cost includes the ressources the partner puts on the development without being supported by the NoE. These efforts must be mesurable at the end of the funding period. Request Support contains the amount of money asked to the NoE. The requested support should precisely specify how it is distributed among salary & equipment.